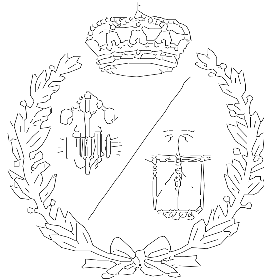
ESCUELA TÉCNICA SUPERIOR DE INGENIEROS
INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA



Proyecto Fin de Grado

APLICACIÓN DE TÉCNICAS DE MACHINE LEARNING PARA LA MEJORA DEL RECONOCIMIENTO ÓPTICO DE CARACTERES (OCR) SOBRE SUPERFICIES INDUSTRIALES

Application of machine learning techniques to improve optical character recognition systems (OCR) on industrial surfaces.

Para acceder al Título de

GRADUADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

Autor: Daniel Roldán Cordero

Febrero – 2023

ÍNDICE DE CONTENIDOS

1. RESUMEN 7
2. ABSTRACT 7
3. ACRÓNIMOS 8
4. MOTIVACIÓN 9
5. INTRODUCCIÓN 9
6. ESTADO DEL ARTE 10
   6.1. Diagrama de flujo de un modelo OCR general 10
      6.1.1. Flujo de entrenamiento 11
      6.1.2. Flujo de test 19
7. MEDICIÓN DE LA CALIDAD DE UN MODELO OCR GENERAL 23
   7.1. Tasas de error y verdad de campo: 23
   7.2. Tipos de errores 23
   7.3. Tasas de error de cálculo 23
8. MEDICIÓN DE LA CALIDAD DE NUESTRO MODELO 26
9. POSIBLES SOLUCIONES 26
10. ESTUDIO PREVIO 27
   10.1. Visión artificial tradicional por computador 27
   10.2. Machine Learning 28
   10.3. Deep Learning 29
   10.4. Conclusión técnica escogida 31
11. DESARROLLO DEL PROGRAMA 32
      11.1. Entradas y salidas 33
      11.2. Estructura 34
      11.3 Funcionamiento 60
12. ELECCIÓN DEL MODELO, PREPARACIÓN DEL DATASET Y ENTRENAMIENTO 70
   12.1. Elección del modelo de Deep Learning 70
   12.2. Preparación del dataset 70
      12.2.1. Script para obtención de recortes 71
   12.3. Tipología de datasets 74
      12.3.1. Códigos sobre superficie de papel 74
      12.3.2. Códigos sobre superficie de metal 75
      12.3.3. Números en 7 segmentos 77

12.3.4. Códigos sobre metal liso 78
12.4. Dataset de entrenamiento 79
12.5. Entrenamiento 81
13. INFERENCIA Y COMPARACIÓN CON OTROS MODELOS 84
13.1. Inferencia en imágenes con superficie metálica 86
13.2. Inferencia en imágenes con superficie de papel 89
13.3. Inferencia en imágenes con superficie de metal lisa 91
13.4. Inferencia sobre pantallas LCD (numeración 7 segmentos) 93
13.5. Inferencia en múltiples imágenes 95
13.5.1. Códigos sobre superficie de metal 95
13.5.2. Códigos sobre superficie de papel: 96
13.5.3. Códigos sobre pantalla LCD: 97
13.5.4. Códigos sobre superficie de metal liso: 98
14. CONCLUSIÓN 99
15. PRÓXIMOS PASOS 99
16. PRESUPUESTO 100
17. BIBLIOGRAFÍA 101

ÍNDICE DE FIGURAS

FIGURA 1. FLUJOGRAMA GENERAL DE UN MODELO OCR. FUENTE: HTTPS://WWW.V7LABS.COM/BLOG/OCR-GUIDE 10
FIGURA 2. EJEMPLO DE BINARIZACIÓN. FUENTE: HTTPS://ES.SLIDESHARE.NET/LONELY113/PROCESAMIENTO-DIGITAL-DE-IMGENES-CON-MATLAB 12
FIGURA 3. EJEMPLO DE CORRECCIÓN DE PERSPECTIVA..........12
FIGURA 4. EJEMPLO DE ELIMINACIÓN DE RUIDO. FUENTE: HTTPS://MANUALESTUTOR.COM/APRENDIZAJE-AUTOMATICO/USO-DE-IA-PARA-ELIMINAR-EL-RUIDO-DE-LOS-PERIODICOS-ESCANEADOS/ 13
FIGURA 5. EJEMPLO DE MEJORA DE CONTRASTE. FUENTE: HTTPS://PDF.WONDERSHARE.ES/EDIT-PDF/EDIT-PDF-CONTRAST.HTML 13
FIGURA 6. EJEMPLO DE THRESHOLDING. FUENTE: HTTPS://SCIKIT-IMAGE.ORG/DOCS/STABLE/AUTO_EXAMPLES/SEGMENTATION/PLOT_THRESHOLDING.HTML 14
FIGURA 7. EJEMPLO DE SEGMENTACIÓN POR CUENCA. FUENTE: HTTPS://WWW.YOUTUBE.COM/WATCH?V=S2WVJPEOUCY................14
FIGURA 8. EJEMPLO DE SEGMENTACIÓN BASADA EN CONTORNOS. FUENTE: HTTP://DANGMINHTHANG.COM/KNOWLEDGE-SHARING/CHARACTERS-SEGMENTATION-AND-RECOGNITION-FOR-VEHICLE-LICENSE-PLATE/ 15
FIGURA 9. EJEMPLO DE SEGMENTADO POR REGIONES. FUENTE: HTTPS://STACKOVERFLOW.COM/QUESTIONS/28591117/HOW-DO-I-SEGMENT-A-DOCUMENT-USING-TESSERACT-THEN-OUTPUT-THE-RESULTING-BOUNDING-B 15
FIGURA 10. EJEMPLO DE SEGMENTACIÓN MORFOLÓGICA. FUENTE: HTTPS://WWW.SEMANTICSCHOLAR.ORG/PAPER/MORPHOLOGICAL-SEGMENTATION-FOR-TEXTURES-AND-VINCENT-DOUGHERTY/EEDD1248CBBF28413AC130942A914338709A9118 16
FIGURA 11. MOMENTOS DE HU. FUENTE: HTTPS://LEARNOPENCV.COM/SHAPE-MATCHING-USING-HU-MOMENTS-C-PYTHON/............17
FIGURA 12. PATRONES TRANSFORMADA DE HOUGH. FUENTE: HTTPS://UNIPYTHON.COM/LA-TRANSFORMADA-LINEA-HOUGH/?UTM_CONTENT=CMP-TRUE 17
FIGURA 13. TIPOLOGÍA DE RESULTADOS. FUENTE: HTTPS://DOCS.AWS.AMAZON.COM/MACHINE-LEARNING/LATEST/DG/MODEL-FIT-UNDERFITTING-VS-OVERFITTING.HTML 18
FIGURA 14. LEVENSHTEIN DEMO. FUENTE: HTTPS://WWW.LET.RUG.NL/~KLEIWEG/LEV/ 24
FIGURA 15. EJEMPLOS DE FOTOGRAFÍA DE ENTRADA PARA ALGORITMOS DE VISIÓN TRADICIONAL. FUENTE: HTTPS://ES.123RF.COM/PHOTO_78645793_ALFABETO-LETRAS-COLECCI%C3%B3N-TEXTO-LETRAS-NEGRAS-CONJUNTO-SOBRE-FONDO-BLANCO-VECTOR-ILLUSTRATION.HTML..........27
FIGURA 16. DATASET DE MNIST. FUENTE: HTTPS://LEARNOPENCV.COM/TAG/MNIST/ 28
FIGURA 17. EJEMPLOS DE IMÁGENES..........29
FIGURA 18. FLUJOGRAMA DE EASYOCR. FUENTE: HTTPS://GITHUB.COM/JAIDEDAI/EASYOCR 31
FIGURA 19: FLUJOGRAMA DE PROCESO. FUENTE: ELABORACIÓN PROPIA. 32
FIGURA 20. EJEMPLOS DE VARIANZAS. FUENTE: HTTPS://DEV.TO/ITMINDS/IMPROVE-YOUR-DEEP-LEARNING-MODELS-WITH-IMAGE-AUGMENTATION-4J7F 33
FIGURA 21. EJEMPLOS DE IMÁGENES DE CÓDIGOS GENERADAS CON NUESTRO PROGRAMA. FUENTE: ELABORACIÓN PROPIA 34
FIGURA 22. PARÁMETROS PARA GENERAR RECORTES. FUENTE: ELABORACIÓN PROPIA 34
FIGURA 23. OFFSET BAJO. FUENTE: ELABORACIÓN PROPIA FIGURA 24. OFFSET ALTO. FUENTE: ELABORACIÓN PROPIA..........35
FIGURA 25. PARÁMETROS PARA LA REALIZACIÓN DE CÓDIGOS. FUENTE: ELABORACIÓN PROPIA 35
FIGURA 26. DICCIONARIO DE REGISTRO DE ELEMENTOS GENERADOS EN LOS CÓDIGOS. FUENTE: ELABORACIÓN PROPIA 36
FIGURA 27. ARCHIVO DE DOCKER: DOCKER-COMPOSE.YML. FUENTE: ELABORACIÓN PROPIA..........36
FIGURA 28. ARCHIVO DOCKERFILE. FUENTE: ELABORACIÓN PROPIA 37
FIGURA 29. ARCHIVO REQUIREMENTS.TXT. FUENTE: ELABORACIÓN PROPIA 37
FIGURA 30. SCRIPT DE CONFIGURACIÓN DE LAS RESPUESTAS DE LA API. FUENTE: ELABORACIÓN PROPIA 38
FIGURA 31. MÉTODO PARA LA GENERACIÓN DEL DATASET DE RECORTES DE NÚMEROS Y LETRAS. FUENTE: ELABORACIÓN PROPIA 38
FIGURA 32. MÉTODO UTILIZADO PARA LA GENERACIÓN DEL DATASET DE CÓDIGOS. FUENTE: ELABORACIÓN PROPIA..........39
FIGURA 33. LIBRERÍAS UTILIZADAS. FUENTE: ELABORACIÓN PROPIA 41
FIGURA 34. INICIO FUNCIÓN PRINCIPAL. FUENTE: ELABORACIÓN PROPIA 41
FIGURA 35. GENERACIÓN DE LISTAS. FUENTE: ELABORACIÓN PROPIA 42
FIGURA 36. CONFIGURACIÓN DE CARPETAS. FUENTE: ELABORACIÓN PROPIA..........42
FIGURA 37. FUNCIONES DE TRATAMIENTO DE DATOS. FUENTE: ELABORACIÓN PROPIA 42

Figura 38. Breve descripción de la función. Fuente: Elaboración propia........43
Figura 39. Bucle principal de la función. Fuente: Elaboración propia........43
Figura 40. Condición letra especial. Fuente: Elaboración propia........44
Figura 41. 45
Figura 42. Offset = 1. Fuente: Elaboración propia Figura 43. Offset = 6. Fuente: Elaboración propia........45
Figura 44. Breve descripción de la función. Fuente: Elaboración propia........45
Figura 45. Función utilizada para la generación de números. Fuente: Elaboración propia........46
Figura 46. Offset = 0. Fuente: Elaboración propia Figura 47. Offset = 6. Fuente: Elaboración propia........46
Figura 48. Script para la generación de logs. Fuente: Elaboración propia........47
Figura 49. Inicio de script principal. Fuente: Elaboración propia........47
Figura 50: Comprobación de directorios. Fuente: Elaboración propia........48
Figura 51. Bucle para tratamiento de imágenes. Fuente: Elaboración propia........48
Figura 52. Generación y guardado del archivo .json de registro. Fuente: Elaboración propia........49
Figura 53. Importación de librerías necesarias. Fuente: Elaboración propia........49
Figura 54. Breve descripción de la función. Fuente: Elaboración propia........50
Figura 55. Declaración de variables necesarias. Fuente: Elaboración propia........50
Figura 56. Bucle para la selección de números que formarán el código. Fuente: Elaboración propia........51
Figura 57 .Bucle para la selección de letras que formarán el código. Fuente: Elaboración propia........51
Figura 58. Logs y mezclado de los códigos. Fuente: Elaboración propia........52
Figura 59. Retornable de la función. Fuente: Elaboración propia........53
Figura 60. Función para pasar de coordenadas absolutas........53
Figura 61. Función que nos leerá el archivo .txt de todas las clases. Fuente: Elaboración propia........54
Figura 62. Breve descripción de la función que realizara la composición de imágenes. Fuente: Elaboración propia........54
Figura 63. Declaración de variables necesarias. Fuente: Elaboración propia........55
Figura 64. Bucle principal de la función **pegar_código.** Fuente: Elaboración propia........56
Figura 65. Condiciones para la posición del código y el borrado de letras en caso de que alguna no entre en la imagen de fondo. Fuente: Elaboración propia........57
Figura 66. Transformado a coordenadas tipo YOLO y pegado de la imagen de la letra / número. Fuente: Elaboración propia........57
Figura 67. Información que se dará por terminal para ver el progreso del programa. Fuente: Elaboración propia........58
Figura 68. Función **guardar_txt** para la generación de etiquetas. Fuente: Elaboración propia........58
Figura 69. Función que genera el diccionario para obtener un registro de las clases generadas. Fuente: Elaboración propia........59
Figura 70. Comando para primer levantamiento de contenedores. Fuente: Elaboración propia........60
Figura 71. Comando para el levantamiento normal de contenedores. Fuente: Elaboración propia........60
Figura 72. Salida por terminal del funcionamiento de la API. Fuente: Elaboración propia........61
Figura 73. URL a la API. Fuente: Elaboración propia........61
Figura 74. Pantalla principal de la API. Fuente: Elaboración propia........61
Figura 75. Pantalla psincipal de la API. Fuente: Elaboración propia........62
Figura 76. Método para la generación del dataset de recortes. Fuente: Elaboración propia........62
Figura 77. Fondo escogido. Fuente: Elaboración propia........63
Figura 78: respuesta de la API al método **dataset_image_parameters.** Fuente: Elaboración propia........63
Figura 79. Carpeta generada con el dataset de los recortes. Fuente: Elaboración propia........63
Figura 80. Ejemplo de imágenes generadas. Fuente: Elaboración propia........64
Figura 81. Método para la generación del dataset de códigos. Fuente: Elaboración propia........64
Figura 82. Respuesta de la API al método **previous_parameters.** Fuente: Elaboración propia........65
Figura 83. Ejemplo de imagen de código generado artificialmente. Fuente: Elaboración propia........66
Figura 84. Etiqueta correspondiente a la imagen anterior. Fuente: Elaboración propia........66
Figura 85. Archivo **classes.txt.** Fuente: Elaboración propia........67

Figura 86. Información del transcurso del programa en imagen normal. Fuente: Elaboración propia. 67
Figura 87. Ejemplo de imagen con borrado de números. Fuente: Elaboración propia. 68
Figura 88. Información del transcurso del programa en imagen con borrado de elementos. Fuente: Elaboración propia. 68
Figura 89. Diccionario de registro de clases. Fuente: Elaboración propia. 69
Figura 90. Importación de librerías del script para la separación de elementos. Fuente: Elaboración propia.................. 71
Figura 91. Declaración de variables principales. Fuente: Elaboración propia. 72
Figura 92. Bucle para recorrer todas las imágenes del dataset. Fuente: Elaboración propia. 72
Figura 93. Iteramos sobre detecciones de YOLO. Fuente: Elaboración propia. 73
Figura 94. Condición que debe de cumplir la detección. Fuente: Elaboración propia.. 73
Figura 95. Dataset de entrada (proyecto de empresa) Figura 96. Dataset de entrada al método **previous_parameters** 74
Figura 97. Dataset de temática (papel en este caso) 74
Figura 98. Imagen de los códigos de los que se extraerán los elementos. Fuente: Elaboración propia. 74
Figura 99. Fondo para dataset de papel. Fuente: Elaboración propia. 75
Figura 100. Imágenes resultado. Fuente: Elaboración propia. 75
Figura 101. Imagen de códigos sobre metal. Fuente: Elaboración propia. 75
Figura 102. Fondo para dataset en metal. Fuente: Elaboración propia. 76
Figura 103. Ejemplos de imágenes de códigos generados en metal. Fuente: Elaboración propia. 76
Figura 104. Ejemplos de variaciones para dataset de 7 segmentos. Fuente: Elaboración propia. 77
Figura 105. Fondo para dataset de números 7 segmentos. Fuente: Elaboración propia. 77
Figura 106. Ejemplos de imágenes generadas de números 7 segmentos. Fuente: Elaboración propia. 78
Figura 107. Ejemplos de imágenes con variaciones para dataset en metal liso. Fuente: Elaboración propia. 78
Figura 108. Fondo para dataset de metal liso. Fuente: Elaboración propia. 78
Figura 109. Ejemplo de imágenes generadas para el dataset de metal liso. Fuente: Elaboración propia........................... 79
Figura 110. Ejemplo de imágenes de los datasets de las distintas superficies. Fuente: Elaboración propia. 79
Figura 111. Distribución de carpetas para el entrenamiento. Fuente: Elaboración propia. 79
Figura 112: Distribución de imágenes y etiquetas para el entrenamiento. Fuente: Elaboración propia. 80
Figura 113. Archivo **train.yaml.** Fuente: Elaboración propia. 80
Figura 114. Parámetros con los que se realizará el entrenamiento. Fuente: Elaboración propia. 81
Figura 115. Matriz de confusión. Fuente: Elaboración propia. 82
Figura 116 Importación de librerías de script para probar EasyOCR. Fuente: Elaboración propia. 84
Figura 117. Programa principal. Fuente: Elaboración propia. 84
Figura 118. Función para procesar imágenes. Fuente: Elaboración propia. 85
Figura 119. Función que nos permitirá observar las detecciones del modelo. Fuente: Elaboración propia...................... 85
Figura 120. Imagen sobre superficie de metal para probar ambos modelos. Fuente: Elaboración propia. 86
Figura 121. Resultados de detección de EasyOCR sobre superficie metálica. Fuente: Elaboración propia. 86
Figura 122. Resultados de detección de nuestro modelo sobre superficie metálica. Fuente: Elaboración propia............. 87
Figura 123. Imágenes de las detecciones de nuestro modelo en formato ".json" sobre superficie metálica. Fuente: Elaboración propia... 88
Figura 124. Imagen sobre superficie de papel para probar ambos modelos. Fuente: Elaboración propia. 89
Figura 125. Resultados de detección de EasyOCR sobre superficie de papel. Fuente: Elaboración propia. 89
Figura 126. Resultados de deteccón de nuestro modelo sobre superficie de papel. Fuente: Elaboración propia............... 90
Figura 127. Imágenes de las detecciones de nuestro modelo en formato ".json" sobre superficie de papel. Fuente: Elaboración propia... 90
Figura 128. Imagen sobre superficie de metal liso para probar ambos modelos. Fuente: Elaboración propia.................. 91
Figura 129. Resultados de detección de EasyOCR sobre superficie de metal liso. Fuente: Elaboración propia.................. 91
Figura 130. Resultados de detección de nuestro modelo sobre superficie metálica lisa. Fuente: Elaboración propia. 92

FIGURA 131. IMÁGENES DE LAS DETECCIONES DE NUESTRO MODELO EN FORMATO ".JSON" SOBRE SUPERFICIE METÁLICA LISA. FUENTE: ELABORACIÓN PROPIA....92
FIGURA 132. IMAGEN SOBRE PANTALLA LCD (NÚMEROS 7 SEGMENTOS) PARA PROBAR AMBOS MODELOS. FUENTE: ELABORACIÓN PROPIA. 93
FIGURA 133. RESULTADO DE DETECCIÓN DE EASYOCR SOBRE PANTALLA LCD. FUENTE: ELABORACIÓN PROPIA....93
FIGURA 134. RESUILTADO DE DETECCÓN DE NUESTRO MODELO SOBRE PANTALLA LCD. FUENTE: ELABORACIÓN PROPIA. 93
FIGURA 135. IMÁGENES DE LAS DETECCIONES DE NUESTRO MODELO EN FORMATO ".JSON" SOBRE PANTALLA LCD. FUENTE: ELABORACIÓN PROPIA. 94
FIGURA 136. ACIERTOS EN DETECCIONES DE EASYOCR PARA SUPERFICIE DE METAL (0/50 ACIERTOS). FUENTE: ELABORACIÓN PROPIA...95
FIGURA 137. ACIERTOS EN DETECCIONES DE NUESTRO MODELO PARA SUPERFICIE DE METAL (46/50 ACIERTOS). FUENTE: ELABORACIÓN PROPIA. 95
FIGURA 138. ACIERTOS EN DETECCIONES DE EASYOCR PARA SUPERFICIE DE PAPEL (42/50 ACIERTOS). FUENTE: ELABORACIÓN PROPIA..96
FIGURA 139. ACIERTOS EN DETECCIONES DE NUESTRO MODELO PARA SUPERFICIE DE PAPEL (50/50 ACIERTOS). FUENTE: ELABORACIÓN PROPIA. 96
FIGURA 140. ACIERTOS EN DETECCIONES DE EASYOCR PARA PANTALLA LCD (35/50 ACIERTOS). FUENTE: ELABORACIÓN PROPIA. 97
FIGURA 141. ACIERTOS EN DETECCIONES DE NUESTRO MODELO PARA PANTALLA LCD (42/50 ACIERTOS). FUENTE: ELABORACIÓN PROPIA. 97
FIGURA 142. ACIERTOS EN DETECCIONES DE EASYOCR PARA SUPERFICIE DE METAL LISO (43/50 ACIERTOS). FUENTE: ELABORACIÓN PROPIA. 98
FIGURA 143. ACIERTOS EN DETECCIONES DE NUESTRO MODELO PARA SUPERFICIE DE METAL LISO (48/50 ACIERTOS). FUENTE: ELABORACIÓN PROPIA....98

1. RESUMEN

El reconocimiento de números y letras grabadas sobre productos comerciales es de vital importancia para que una industria pueda tener la trazabilidad e identificación de sus productos.

A pesar de ser esencial, el reconocimiento y la clasificación de forma automática es un problema aun sin resolver ya que existen infinidad de superficies y métodos de grabado que presentan, a su vez, grandes variaciones, siendo un reto para los algoritmos de reconocimiento por imagen que tenemos actualmente.

Por tanto, para solucionar esto, en el presente proyecto vamos a abordar un modelo de reconocimiento de imágenes mediante técnicas de Deep Learning, que detecte y clasifique códigos alfanuméricos con alta precisión, independientemente de la superficie y de la técnica de grabado que se utilice.

Como la solución de utilizar Deep Learning utiliza grandes cantidades de datos (imágenes en nuestro caso) vamos a generar un **dataset** que recoja cientos de imágenes de distintas superficies y acabados, que uniremos aleatoriamente en miles de combinaciones.

Esto permitirá tener suficientes datos de la distribución con el fin de entender, de forma objetiva y medible cuales son los puntos de trabajo ideales y los casos más complejos a estudiar con el fin de establecer un camino claro para futuros avances.

2. ABSTRACT

The recognition of numbers and letters engraved on commercial products is of vital importance so that an industry can have the traceability and identification of its products.

Despite being essential, automatic recognition and classification is an unresolved problem as there are countless surfaces and engraving methods that present, in turn, great variations, being a challenge for image recognition algorithms. that we currently have.

Therefore, to solve this, in this project we are going to address an image recognition model using Deep Learning techniques, which detects and classifies alphanumeric codes with high precision, regardless of the surface and the engraving technique used.

As the solution to use Deep Learning uses large amounts of data (images in our case) we are going to generate a dataset that collects hundreds of images of different surfaces and finishes, which we will randomly join in thousands of combinations.

This will allow having enough distribution data in order to understand, in an objective and measurable way, which are the ideal working points and the most complex cases to study in order to establish a clear path for future advances.

3. ACRÓNIMOS

- **Dataset**: conjunto de datos (imágenes en nuestro caso) con los que el modelo va a realizar el entrenamiento.
  - Son una parte vital para el desarrollo de cualquier modelo de inteligencia artificial.
  - Estos conjuntos de datos pueden consistir en imágenes, videos, texto, audio, o cualquier otro tipo de información que se pueda utilizar para entrenar un modelo.
  - Es muy importante que el dataset sean representativos y estén bien etiquetados para que el modelo pueda aprender de manera eficaz.
- **Bounding box** (bbox): caja delimitadora que engloba un elemento (números y letras en nuestro caso).
- **Etiquetas**: archivo .txt con el registro de las clases de una imagen, el centro "x" e "y" de las bbox, así como el ancho y el alto de esta.
- **YOLO**: modelo de Deep Learning basado en redes neuronales convolucionales (CNN), especializado en aprender patrones en imágenes y realizar predicciones. En este proyecto se utilizará este algoritmo en su versión 8.
- **Overfitting** de un modelo: se podría decir que se tata del sobreaprendizaje del modelo, es decir, este se aprende los datos de entrenamiento, y no es capaz de generalizar cuando trabaje con datos distintos a los datos con los que se entrenó.
- **Inferencia:** proceso de utilización del modelo entrenado para hacer predicciones en nuevos datos de entrada, sin realizar actualizaciones de los parámetros y en función de la arquitectura y los pesos preaprendidos.
- **Pesos:** los parámetros aprendidos por un modelo durante el entrenamiento, que capturan la información y el conocimiento extraídos de los datos para realizar predicciones en la etapa de inferencia.
- **API**: Interfaz gráfica utilizada para una mejor y más sencilla interacción con el programa. En nuestro caso se utilizará la librería **FASTAPI** de Python**.**
- **Tiling**: técnica que consiste en colocar múltiples imágenes en una disposición repetitiva, creando un patrón o mosaico, en la que cada imagen se coloca contigua a las otras, de manera que se crea una apariencia continua y uniforme.

4. MOTIVACIÓN

El principal motor a la hora de decidir realizar un trabajo de fin de grado sobre tecnología OCR aplicada a la industria, es debido a que cada día vemos una mayor demanda por parte de las industrias para poder automatizar lo máximo posible sus procesos productivos. Sin embargo, aunque esta tecnología este experimentando una mejora continua, sigue teniendo una precisión limitada a la hora de trabajar con datos complicados (caracteres o números parcialmente borrados o dañados) lo que genera falsos rechazos.

Por tanto, pensamos que crear un modelo lo más generalista y preciso posible es de vital importancia para que la tecnología OCR entre aún más en el ámbito industrial y ayude a las empresas a ahorrar costes.

5. INTRODUCCIÓN

La tecnología OCR (Reconocimiento Óptico de Caracteres) se ha convertido en una herramienta esencial en la industria gracias a su capacidad para automatizar la lectura y procesamiento de datos. Esta tecnología permite la digitalización de documentos, detección de caracteres, trazabilidad…, lo que reduce los tiempos de proceso y aumenta la eficiencia en una variedad de sectores industriales.

En este trabajo de fin de grado, se analizará el uso de la tecnología OCR en industria, con énfasis en su aplicación en la automatización de procesos, mejora de la calidad y la reducción de costes de fabricación. Se discutirán también las limitaciones y desafíos actuales en el uso de la tecnología OCR y se propondrán soluciones para superarlos.

- En la industria, la tecnología OCR se utiliza principalmente para automatizar la lectura y procesamiento de documentos, como facturas, recibos y formularios, lo que ayudará a las empresas reducir los tiempos de procesamiento y mejorar la precisión de los datos.

- En el sector financiero, la tecnología OCR se utiliza para automatizar el procesamiento de facturas y recibos, lo que ha permitido a las empresas ahorrar costes económicos y temporales.
  En este sector también es importante destacar que la mayoría de estos documentos eran escritos a mano, por lo que la implantación de esta tecnología ahorrará tiempo en los procesos.

- En logística, esta técnica se utiliza para automatizar la lectura de códigos de barras, permitiendo una mayor precisión en la gestión de inventarios y mejorar la trazabilidad.

6. ESTADO DEL ARTE

La tecnología OCR se hizo popular a principios de la década de 1990 mientras se digitalizaban periódicos. Desde sus inicios, la tecnología ha experimentado multitud de mejoras.

Las soluciones actuales tienen la capacidad de ofrecer una precisión de OCR cercanas a las humanas, pero con tiempos de ciclo muy reducidos. Se utilizan métodos avanzados para automatizar flujos de trabajo de procesamiento de documentos complejos.

Antes de que la tecnología OCR estuviera disponible, la única opción para formatear documentos digitalmente era volver a escribir manualmente el texto. Esto no solo consumía mucho tiempo, sino que también venía con inexactitudes inevitables y errores. Hoy en día, los servicios de OCR están ampliamente disponibles para el público.

6.1. Diagrama de flujo de un modelo OCR general

A continuación, se muestra el diagrama de flujo de un modelo de OCR:

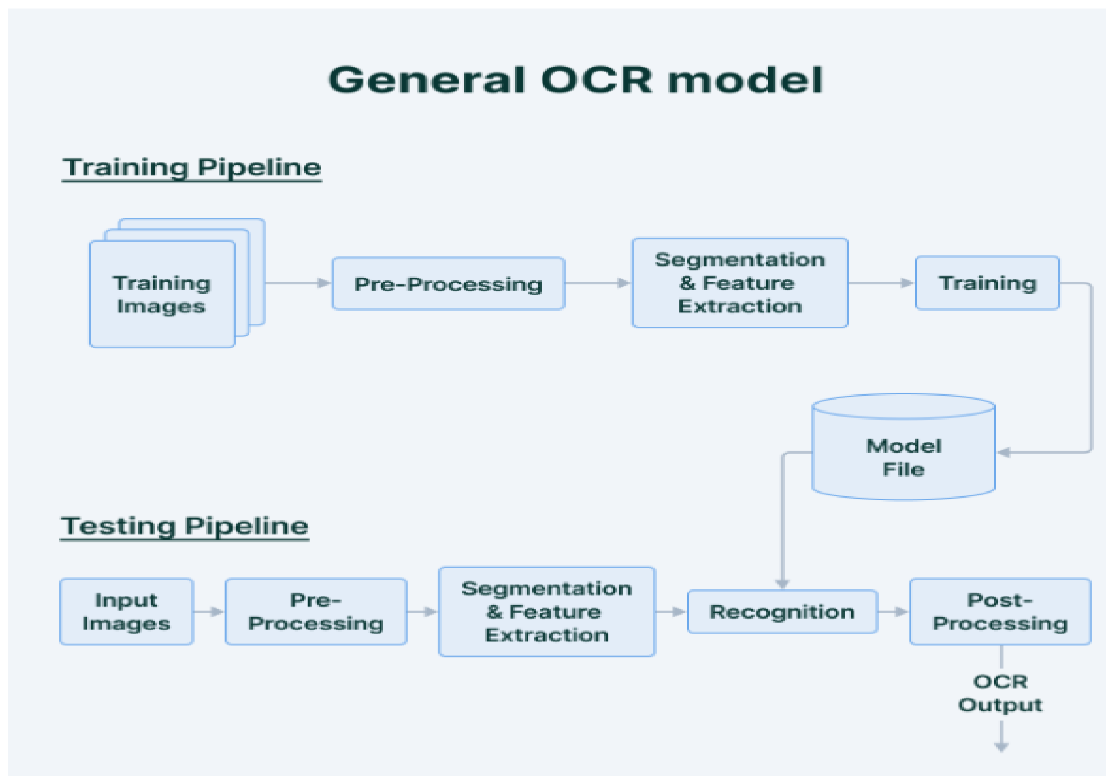


Figura 1. Flujograma general de un modelo OCR. Fuente: https://www.v7labs.com/blog/ocr-guide

Como se puede observar, el proceso de ejecución de un modelo de OCR se podría dividir en dos zonas:

6.1.1. Flujo de entrenamiento

El entrenamiento de un modelo consiste en alimentarlo con un conjunto de datos de entrenamiento y ajustar los parámetros del modelo para que pueda hacer predicciones precisas y generalizadas en nuevos datos.

La finalidad de esta etapa es tener un modelo lo más robusto posible ante la aparición de variaciones en el entorno de trabajo

a. Datos destinados a entrenamiento

Conjunto de datos (imágenes en nuestro caso) junto con sus correspondientes etiquetas que nos ayudarán a que el modelo aprenda de la mejor manera posible.

Es de vital importancia tener unos datos equilibrados para el entrenamiento ya que los resultados van a depender en gran medida de la calidad de estos.

Si en el conjunto de datos estas varianzas no aparecen o están desbalanceadas, el entrenamiento no va a ser optimo y el modelo fallara en la etapa de test.

En cambio, tampoco tienen que aparecer todas las varianzas posibles (principalmente porque es imposible abarcar todas las posibilidades de cambio de un entorno).

En cambio, tampoco podemos permitir que el modelo aprenda patrones que son específicos de los datos de entrenamiento ya que esto daría lugar a un problema llamado **overfitting.**

El **overfitting s**e produce cuando un modelo se ajusta demasiado a los datos de entrenamiento y pierde su capacidad de generalización en datos nuevos o desconocidos. Es decir, el modelo se aprende las imágenes sin tener o teniendo muy poca capacidad posteriormente de trabajar con imágenes diferentes a las del entrenamiento haciendo que los resultados del entrenamiento sean excelentes, pero que en la etapa de test el modelo falle repetidamente.

Para evitar esto, hemos diseñado un generador de datos que hará combinaciones de números y letras aleatorias procedentes de otros dataset y las unirá en una imagen cuyo fondo podemos escoger, este algoritmo se explicará posteriormente en el apartado de desarrollo.

b. Preprocesado

El preprocesamiento de una imagen es el conjunto de técnicas que se le aplican a esta para intentar que quede en unas condiciones lo más ideales posible para que el entrenamiento sea más rápido y podamos tener unos mejores resultados.

Esta es una etapa muy importante del proceso ya que cuanto mejor se “idealice” una imagen (sin llegar a cambiar sus propiedades) mejor será el modelo.

Dentro de esta etapa se podrían aplicar técnicas como:

- **Binarización**: La binarización es una técnica que convierte una imagen en escala de grises a una imagen en blanco y negro (0 y 1). Esto se hace para separar los caracteres del fondo y eliminar el ruido de que pueda dificultar la lectura del OCR. Existen varios métodos de binarización, como la umbralización global (thresholding), que utiliza un solo valor de umbral para toda la imagen, y la umbralización adaptativa (adaptive thresholding), que ajusta el valor de umbral en función de las características locales de la imagen.

Figura 2. Ejemplo de binarización. Fuente: https://es.slideshare.net/lonely113/procesamiento-digital-de-imgenes-con-matlab

- **Corrección de perspectiva**: Si la imagen está inclinada o tiene una perspectiva irregular, la corrección de perspectiva se utiliza para enderezar la imagen. Esto se hace para que los caracteres aparezcan en una posición más uniforme, lo que facilita la lectura del modelo. Una técnica muy utilizada aplicada para corregir la perspectiva es la transformación de Hough, que detecta las líneas rectas en la imagen y calcula la rotación necesaria para alinearlas verticalmente.

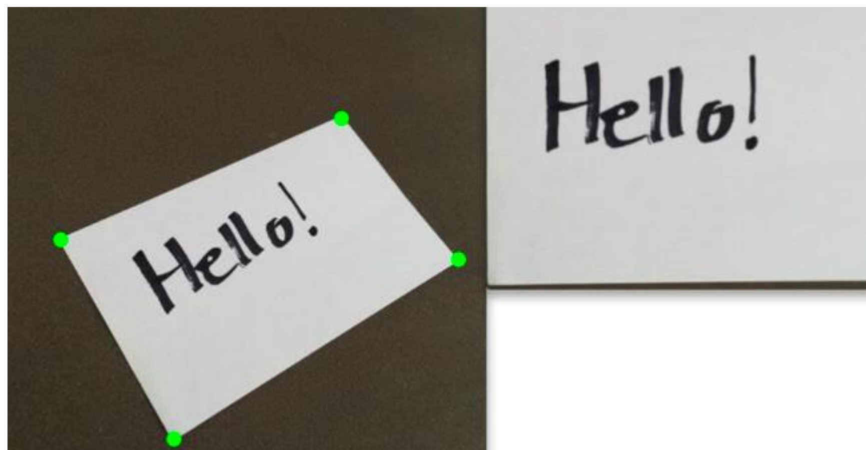


Figura 3. Ejemplo de corrección de perspectiva

- **Eliminación de ruido**: Esta técnica se utiliza para reducir las manchas y los puntos no deseados en la imagen. El ruido puede ser generado por varias causas, como la calidad del escaneo o la calidad de la imagen original. La eliminación de ruido se puede hacer mediante filtros, como el filtro de mediana o el filtro Gaussiano, que eliminan el ruido manteniendo los detalles importantes de la imagen.

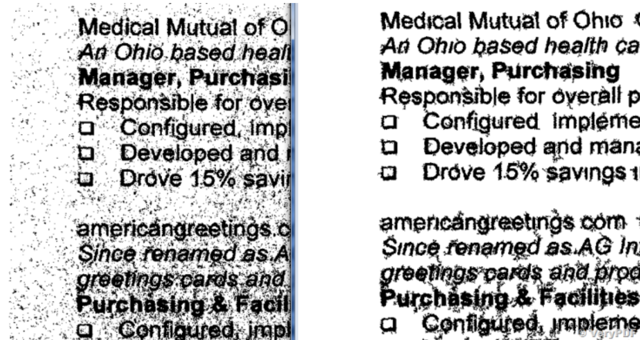


Figura 4. Ejemplo de eliminación de ruido. Fuente: https://manualestutor.com/aprendizaje-automatico/uso-de-ia-para-eliminar-el-ruido-de-los-periodicos-escaneados/

- **Mejora de contraste**: Técnica similar a la binarización, la mejora de contraste se utiliza para ajustar el contraste de la imagen y mejorar la visibilidad de los caracteres. Se puede llevar a cabo mediante técnicas como el ajuste automático de contraste, que utiliza algoritmos para ajustar el contraste de la imagen de manera automática o manualmente.

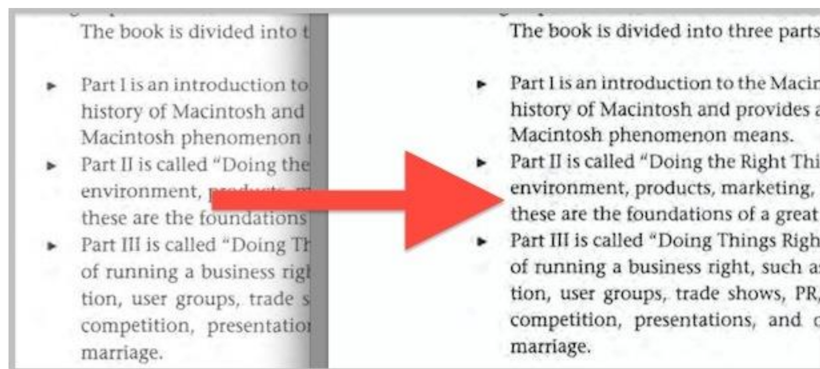


Figura 5. Ejemplo de mejora de contraste. Fuente: https://pdf.wondershare.es/edit-pdf/edit-pdf-contrast.html

- **Eliminación de bordes**: La eliminación de bordes se utiliza para eliminar los bordes de la imagen que no son relevantes para la lectura del OCR. Se puede llevar a cabo mediante técnicas como el recorte, que elimina una sección de la imagen, o la extracción de contenido, que elimina automáticamente el **contenido no deseado.**

- **Suavizado**: El suavizado se utiliza para suavizar los bordes de los caracteres y mejorar su legibilidad para el OCR. Es fácilmente regulable dependiendo de las características de cada imagen.

c. Segmentación y extracción de características

La **segmentación** es el proceso de dividir una imagen de texto en diferentes regiones, cada una de las cuales contiene un solo carácter. Esto es necesario para que el modelo pueda identificar cada carácter individualmente.

La segmentación se realiza mediante técnicas de procesamiento de imágenes, que buscan contornos o bordes de los caracteres y separan cada uno de ellos en una región independiente.

Algunos de los algoritmos usados podrían ser:

- **Thresholding (Umbralización)**: Esta técnica convierte una imagen en una imagen binaria, donde los píxeles por encima de un umbral dado se clasifican como blanco y los píxeles por debajo del umbral se clasifican como negro. Es útil para separar caracteres de fondo y hacer una diferenciación en elementos que se sitúen a poca distancia.

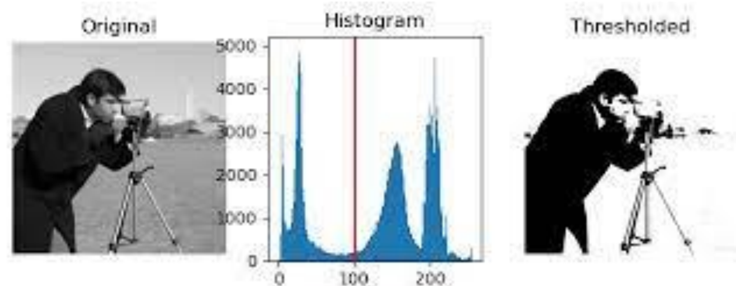


Figura 6. Ejemplo de thresholding. Fuente: https://scikit-image.org/docs/stable/auto_examples/segmentation/plot_thresholding.html

- **Watershed Segmentation (Segmentación por cuenca)**: Técnica basada en la idea de la cuenca hidrográfica. Se separan los elementos como si fueran picos en una cuenca, es muy útil para segmentar caracteres con líneas de base curvas y documentos de texto como libros, manuscritos, etc...

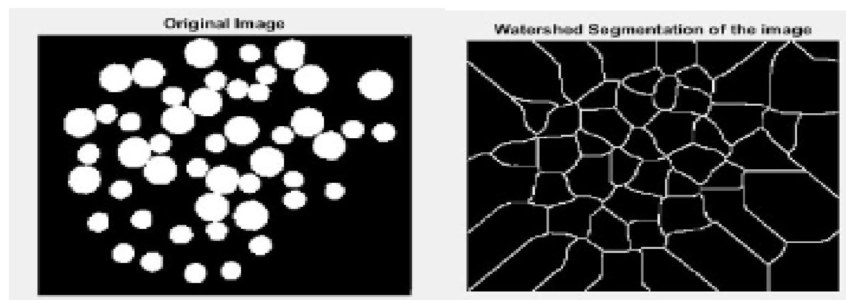


Figura 7. Ejemplo de segmentación por cuenca. Fuente: https://www.youtube.com/watch?v=s2wVJpeoUcY

- **Contour-based Segmentation (Segmentación basada en contornos)**: Esta técnica encuentra los contornos de los caracteres y los separa del fondo, se utiliza cuando los caracteres a analizar tienen un contorno claro y bien definido, por ejemplo, una imagen en la que los caracteres sean blancos y el fondo negro como la que se observa a continuación.

Figura 8. Ejemplo de segmentación basada en contornos. Fuente: http://dangminhthang.com/knowledge-sharing/characters-segmentation-and-recognition-for-vehicle-license-plate/

- **Region-based Segmentation (Segmentación basada en regiones)**: Esta técnica divide la imagen en regiones basadas en la textura, el color o la intensidad de los píxeles. Se suele utilizar para imágenes con regiones de caracteres bien definidas.

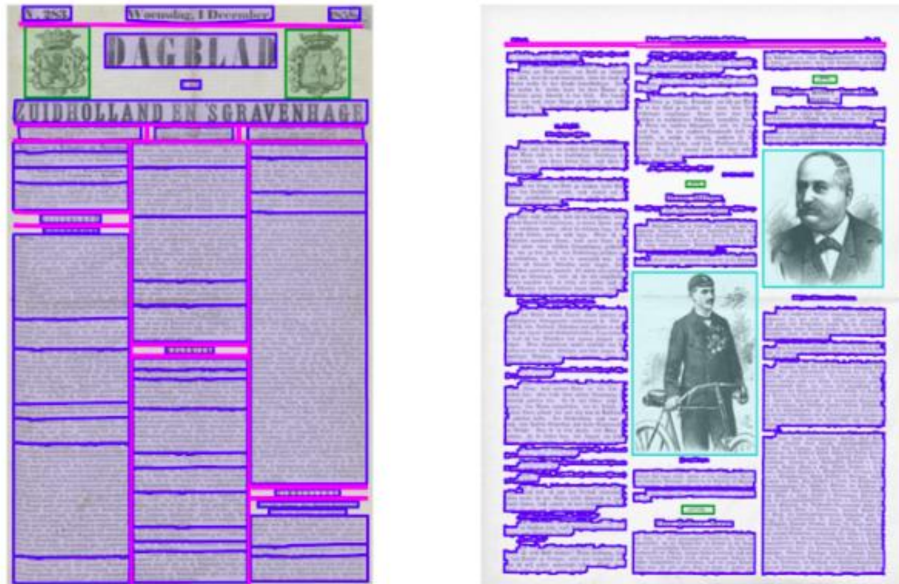


Figura 9. Ejemplo de segmentado por regiones. Fuente: https://stackoverflow.com/questions/28591117/how-do-i-segment-a-document-using-tesseract-then-output-the-resulting-bounding-b

- **Morphological Segmentation (Segmentación morfológica)**: Esta técnica utiliza operaciones morfológicas como **dilatación** y **erosión** para separar los caracteres de la imagen de fondo. Es muy útil para imágenes con caracteres que entran en contacto o se superpongan.

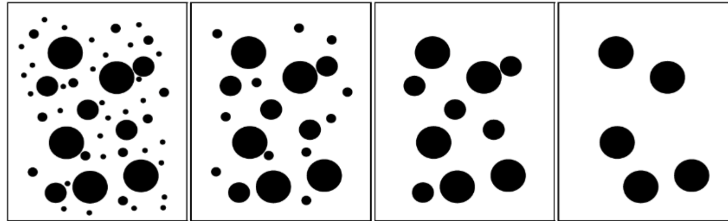


Figura 10. Ejemplo de segmentación morfológica. Fuente: https://www.semanticscholar.org/paper/Morphological-Segmentation-for-Textures-and-Vincent-Dougherty/eedd1248cbbf28413ac130942a914338709a9118

La **extracción de características** se realiza una vez acabada la segmentación. En esta etapa, se extraen las características relevantes de cada uno de los caracteres segmentados como pueden ser la forma, el tamaño, la inclinación, etc...

Este proceso se realiza mediante algoritmos de procesamiento de imágenes y aprendizaje automático, que buscan patrones en los datos y los utilizan para crear modelos que puedan reconocer los caracteres.

El principal **objetivo** de la extracción de características es obtener la imagen en unas condiciones lo más ideales posibles para que se pueda realizar después el reconocimiento correctamente.

Algunos de los algoritmos usados podrían ser:

- **Redes neuronales convolucionales (CNN):** Estas redes son un tipo de algoritmo de aprendizaje profundo que se utilizan comúnmente en la extracción de características para OCR, utilizan capas convolucionales para identificar características en la imagen, como bordes, texturas y formas que posteriormente se utilizan para crear un modelo que pueda reconocer los caracteres en la imagen.

- **Momentos de Hu:** Los momentos de Hu son un conjunto de características geométricas utilizadas en el análisis de imágenes. Estos momentos describen la forma y la orientación de un objeto en una imagen se suelen utilizar en la extracción de características de caracteres OCR gracias a su capacidad para describir la forma de un carácter de manera única.

id	Image	H[0]	H[1]	H[2]	H[3]	H[4]	H[5]	H[6]
K0	K	2.78871	6.50638	9.44249	9.84018	-19.593	-13.1205	19.6797
S0	S	2.67431	5.77446	9.90311	11.0016	-21.4722	-14.1102	22.0012
S1	S	2.67431	5.77446	9.90311	11.0016	-21.4722	-14.1102	22.0012
S2	S	2.65884	5.7358	9.66822	10.7427	-20.9914	-13.8694	21.3202
S3	S	2.66083	5.745	9.80616	10.8859	-21.2468	-13.9653	21.8214
S4	S	2.66083	5.745	9.80616	10.8859	-21.2468	-13.9653	-21.8214

Figura 11. Momentos de HU. Fuente: https://learnopencv.com/shape-matching-using-hu-moments-c-python/

- **Transformada de Hough:** La transformada de Hough es un algoritmo utilizado para detectar formas y patrones en una imagen. En el ámbito de OCR, se utiliza comúnmente para detectar la inclinación de los caracteres y corregir su orientación para mejorar la precisión del reconocimiento.

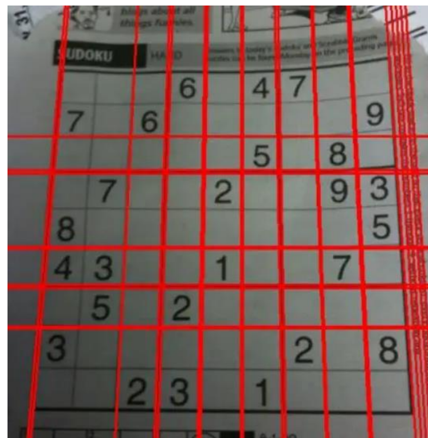


Figura 12. Patrones transformada de Hough. Fuente: https://unipython.com/la-transformada-linea-hough/?utm_content=cmp-true

d. Entrenamiento

El entrenamiento del modelo es la última etapa de esta zona, pero no por ello la menos importante. Esta etapa se puede organizar en los siguientes pasos:

1. **Elección del modelo**: El primer paso en el entrenamiento de un modelo de OCR es elegir el modelo adecuado.

2. **Etiquetado de imágenes:** Es un proceso algo costoso ya que se invierte una considerable cantidad de tiempo (a más imágenes mayor tiempo de etiquetado), pero muy importante para el correcto funcionamiento del modelo. En este proceso iremos mirando imagen a imagen y etiquetando los objetos que deseamos detectar gracias a una herramienta de etiquetado como puede ser: **LabelStudio** o **LabelImg** entre otras.

3. **Entrenamiento del modelo**: El siguiente paso es entrenar el modelo para que aprenda a reconocer los diferentes caracteres y las variaciones en las fuentes y tamaños.

   Como **resultado** del entrenamiento obtendremos los **pesos**.

4. **Validación del modelo**: Después de entrenar el modelo, se debe validar su precisión utilizando datos de validación que no se utilizaron en el entrenamiento. Esto permite evaluar la capacidad del modelo para generalizar y predecir el texto de entrada.

5. **Ajuste del modelo**: Si el modelo no cumple con las expectativas, se ajusta y se vuelve a entrenar utilizando diferentes parámetros y técnicas hasta que se obtiene una precisión aceptable**.** Las variables que permiten este ajuste se denominan **hiperparámetros.**

6. **Interpretación de resultados**: Finalmente, después de entrenar y ajustar el modelo, se utiliza para predecir el texto de entrada en imágenes de prueba, esta interpretación implica la comparación del texto de salida del modelo con el texto real y la evaluación de la precisión del modelo en la predicción del texto. Si se necesitan mejoras en la precisión, se pueden realizar ajustes adicionales en el modelo.

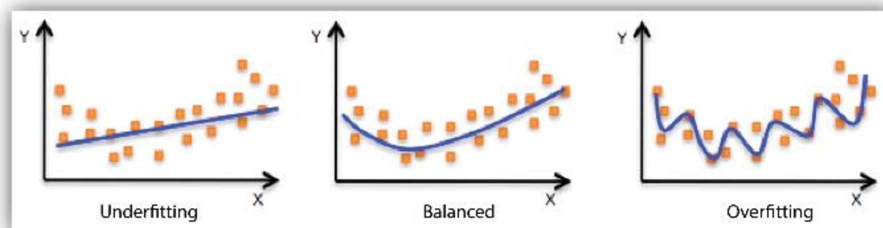


Figura 13. Tipología de resultados. Fuente: https://docs.aws.amazon.com/machine-learning/latest/dg/model-fit-underfitting-vs-overfitting.html

Como se observa en la primera gráfica, se hace referencia al subajuste del modelo, lo que quiere decir que el modelo es demasiado simple y no logra capturar las complejidades de los datos de entrenamiento. Esto se puede deber a falta de imágenes, etiquetas erróneas...

En la segunda gráfica, observamos que el modelo encuentra un equilibrio óptimo entre la capacidad de generalización y el ajuste a los datos.

En la tercera gráfica, el modelo hace un sobreajuste en los datos de entrenamiento perdiendo la capacidad de generalización en nuevos datos.

Posteriormente se explicará la fase de entrenamiento particularizada para nuestro caso.

6.1.2. Flujo de test

El objetivo de evaluar un modelo es medir su rendimiento y determinar su capacidad para hacer predicciones precisas en situaciones del mundo real

a. Datos de test

Conjunto de datos (imágenes en nuestro caso) que se le pasan al modelo entrenado para realizar las detecciones correspondientes gracias a los **pesos** generados durante el entrenamiento.

b. Preprocesado

Igual al preprocesado descrito en el apartado “b” de la zona de entrenamiento, pero aplicándose a las imágenes que se le pasan a modelo para el test.

c. Segmentación y extracción de características

Igual a la segmentación y extracción de características descrita en el apartado “c” de la zona de entrenamiento, pero aplicándose a las imágenes que se le pasan a modelo para el test.

d. Reconocimiento

Este proceso implica utilizar un algoritmo de reconocimiento de patrones para comparar las características extraídas de cada carácter con las de un conjunto de caracteres conocidos previamente.

En este apartado haremos uso de los pesos obtenidos en la zona de entrenamiento para que el modelo pueda reconocer los distintos patrones de las letras.

Una vez tenemos la imagen en las condiciones más ideales posibles, se aplicarán las siguientes técnicas:

1. **Comparación de características**: Se realiza la comparación con las características de los caracteres conocidos previamente. El algoritmo de reconocimiento de patrones puede utilizar técnicas como la correlación cruzada, el análisis discriminante, las redes neuronales artificiales, entre otras.

    - Correlación cruzada: Esta técnica compara la imagen del carácter de entrada con cada uno de los caracteres conocidos previamente, la comparación se realiza utilizando una métrica de similitud, como la correlación cruzada, que mide el grado de coincidencia entre las características de los dos caracteres.

      El carácter conocido previamente que tenga la mayor similitud con el carácter de entrada se clasifica como el carácter reconocido.

      Su principal **ventaja** es su capacidad de medir la similitud entre dos imágenes a pesar de las posibles variaciones en la escala, la rotación, la iluminación y otros factores de transformación.

      Alunas de sus **desventajas** son la sensibilidad a los ruidos y la necesidad de ajustar cuidadosamente los parámetros de la técnica para obtener resultados precisos.

    - Análisis discriminante: Este análisis se utiliza un modelo de aprendizaje automático.

      Este modelo se entrena con un conjunto de caracteres conocidos previamente y sus características, y se utiliza para clasificar el carácter de entrada en una de las clases. El modelo puede ser basado en métodos estadísticos, como el **análisis discriminante lineal** o el **análisis discriminante cuadrático**.

    - Redes neuronales artificiales: Esta técnica utiliza una red neuronal artificial para realizar la clasificación del carácter de entrada.

      La red neuronal se entrena con un conjunto de caracteres conocidos previamente y sus características, y se utiliza para clasificar el carácter de entrada en una de las clases en función de sus características, comúnmente se utilizan las **redes neuronales convolucionales (CNN).**

    - Modelos de Márkov ocultos: Esta técnica utiliza un modelo de Márkov oculto (HMM) para modelar la secuencia de caracteres en una palabra o una línea de texto, el modelo se entrena con un conjunto de palabras o líneas de texto conocidas previamente, y se utiliza para estimar la probabilidad de que la secuencia de caracteres de entrada forme parte de una palabra o una línea de texto.
      Utilizando algoritmos de entrenamiento, los HMM aprenden a reconocer patrones para mejorar la precisión de la transcripción automática de texto.

2. **Clasificación del carácte**r: Finalmente, se clasifica el carácter de la imagen como el carácter conocido o desconocido.

    - Visión tradicional:
      Si se encuentra una coincidencia cercana, se clasifica el carácter de la imagen como el carácter conocido. Si no se encuentra una coincidencia cercana, el carácter se clasifica como desconocido.

      Una técnica que se suele utilizar es la **distancia euclidiana** que es una medida de la distancia geométrica entre dos puntos en un espacio n-dimensional.

      La fórmula para calcular la **distancia euclidiana** entre dos puntos en un espacio bidimensional (x, y) es la siguiente:

      $$d = sqrt((x2 - x1)^2 + (y2 - y1)^2)$$

      En el caso de espacios n-dimensionales, la fórmula se generaliza como:

      $$d = sqrt((x2 - x1)^2 + (y2 - y1)^2 + \ldots + (z_n - z_{n-1})^2)$$

      Donde x1, y1, z1... representan las coordenadas del primer punto, y x2, y2, z2... representan las coordenadas del segundo punto.

      Por ejemplo, si se tiene un vector de características que representa la altura, ancho, inclinación y forma del carácter "A", y se quiere comparar este vector con un vector de características que representa la misma información para el carácter "B", se puede calcular la distancia euclidiana entre estos dos vectores de características para determinar su similitud.

      En general, cuanto menor sea la distancia euclidiana entre dos vectores de características, mayor será su similitud. Por lo tanto, la distancia euclidiana es una medida útil para la comparación de características en el reconocimiento de caracteres del OCR.

    - Deep Learning o Marchine Learning:
      La clasificación de los caracteres se realiza mediante al reconocimiento de patrones gracias al aprendizaje automático, es decir, por poner un ejemplo, si el modelo ha sido entrenado con muchos números”1” de distintas tipologías, cuando se encuentre con un “1” que no ha visto, buscara la máxima similitud de los patrones de este número y los comparara con los patrones de los números con los que ha sido entrenado.
      Dependiendo de si resulta que los patrones son muy parecidos o no, el modelo detectará ese “1” con una confianza alta (confianza>=0.7) o baja (confianza<0.7)

e. Posprocesado

El posprocesado mayoritariamente se utiliza para corregir en la medida de lo posible los errores generados durante el reconocimiento, para que la salida sea lo más ideal posible.

Se pueden aplicar varias técnicas de corrección de errores como:

- **Segmentación de líneas y palabras**: División del texto reconocido en líneas y palabras individuales. Una segmentación precisa puede mejorar significativamente la calidad del resultado final.

- **Detección y corrección de errores de formato**: Consiste en identificar y eliminar errores de formato, como el tamaño y la fuente de letra, la orientación del texto y la presencia de caracteres en mayúscula o minúscula.

- **Corrección de errores de reconocimiento**: Proceso de eliminado de errores de reconocimiento del modelo de OCR. Puede involucrar el uso de técnicas de comparación de palabras y diccionarios, que permiten identificar y corregir palabras o caracteres que no coinciden con el diccionario o el contexto del texto.

f. Salida

La salida de un modelo de OCR es un texto que representa la transcripción del contenido de la imagen de entrada, es decir, el modelo de OCR recibe una imagen de un texto o conjunto de números y produce una versión digital del contenido de dicha imagen.

Es importante destacar que la calidad de la salida del modelo de OCR dependerá de la calidad de la imagen de entrada y de la precisión del modelo de OCR utilizado.

7. MEDICIÓN DE LA CALIDAD DE UN MODELO OCR GENERAL

Como hemos visto anteriormente, la salida de un modelo de OCR puede no ser perfecta y contener ciertos errores de reconocimiento, por tanto, es de vital importancia tratar de contabilizar la calidad de estos modelos en función de los fallos que se pueden generar.

La calidad de un OCR se puede medir por los siguientes parámetros:

7.1. Tasas de error y verdad de campo:

A menudo, la salida de un algoritmo de OCR contiene errores como palabras mal escritas o caracteres falsos. Para tener una medida independiente a la longitud del texto, se normaliza la longitud del contenido esperado.
Al cociente entre en número de errores y la longitud del texto se le llama **tasa de error.**

7.2. Tipos de errores

- Errores de inserción (símbolos espurios)
- Errores de eliminación (texto perdido o caracteres borrados)
- Errores de sustitución (caracteres incorrectamente escritos).

7.3. Tasas de error de cálculo

Para calcular la tasa de error de un OCR hay que tener en cuenta las siguientes cuestiones:

1. Número mínimo de errores

Calcular la tasa de error no es una tarea sencilla cuando hay muchos errores en la detección del texto.

Para ello primero tenemos que saber que es una **subsecuencia**; una subsecuencia es cualquier cadena de caracteres que se pueda obtener eliminando algún carácter de la cadena original, por ejemplo, la palabra "ola" (salida), sería una subsecuencia de la palabra "hola" (referencia), otra subsecuencia únicamente con las consonantes podría ser "hl".

Para tratar la similitud entre la palabra referencia y su subsecuencia vamos a hacer uso de una técnica, la CER.

- **CER** o tasa de error de caracteres: se calcula como el número mínimo de operaciones necesarias para transformar el texto de referencia en el resultado, este número se conoce como **"Distancia de Levenshtein"** y cuanto menor sea este número, más similitud habrá entre las dos palabras.

$$CER = (i + s + d) / n$$

i. número mínimo de inserciones de caracteres

s: número mínimo de sustituciones

d: número mínimo de eliminaciones

n: número total de caracteres

Sin embargo, calcular esta distancia no es sencillo ya que a menudo la salida puede verse afectada por el ruido y otros factores que influyan en la calidad del texto de entrada.

A continuación, se muestra un ejemplo de la demo para el cálculo de la distancia de Levenshtein.

- **WER** o tasa de error de palabras: se encuentra un nivel por encima de la CER y es prácticamente igual, pero la WER es aplicada a palabras en vez de únicamente a caracteres.

$$WER = (i_W + s_W + d_W) / n_W$$

i_W. número mínimo de inserciones de palabras

s_W: número mínimo de sustituciones

d_W: número mínimo de eliminaciones

n_W: número total de palabras

Figura 14. Levenshtein demo. Fuente: https://www.let.rug.nl/~kleiweg/lev/

2. Normalización

La normalización es una parte esencial ya que el número de errores puede ser mayor que la longitud del texto de referencia dando lugar a índices mayores del 100%, por tanto, el número de errores se divide por la suma del número de operaciones de edición (i + s + d) descritas anteriormente y el número “c” de símbolos correctos, que siempre es mayor que el numerador (i + s + d + c).

3. Espacios en blanco

   Los espacios en blanco son importantes en cualquier texto ya que separan las palabras y deben tenerse en cuenta al calcular la tasa de error de caracteres.

   Es importante señalar que el espacio en blanco no es un carácter como los demás: cuando la alineación entre los textos permite reemplazar el espacio en blanco con un carácter imprimible, a veces conduce a resultados extraños.

   Por ejemplo, al comparar "bad man" y "batman", la eliminación de la d y la sustitución del espacio en blanco por la “t” tiene un coste de Levenshtein idéntico (dos operaciones) que la eliminación del espacio en blanco más la sustitución de la “d” por la “t” (también dos operaciones) pero la segunda opción parece una transformación más natural.

   Este comportamiento se puede minimizar si no se permite la sustitución de espacios en blanco con caracteres imprimibles, esto es de vital importancia ya que a menudo lleva a la generación de errores.

4. Mayúsculas

   Que las letras sean mayúsculas o minúsculas a la hora de analizar un texto es muy importante ya que, una palabra o conjunto de palabras iguales, a excepción de alguna letra minúscula convertida en mayúscula puede cambiar completamente su significado.

   Por ejemplo: "Guardo" (nombre de población) y "guardo” (presente simple del verbo guardar)

   Esto sugiere que la **CER** no debería contar como errores la sustitución de una letra mayúscula por la letra minúscula correspondiente.

5. Codificación de caracteres

   Desafortunadamente, existe una gran cantidad de codificaciones alternativas para archivos de texto, sin embargo, el estándar de codificación de texto más usado es **Unicode**, este sistema asigna a cada carácter un numero entero único (su correspondiente código), con el fin de representar los caracteres que no se pueden representar con **ASCII**.

   Debido a que los mapas estándar superan los 110.000 caracteres, los códigos se almacenan en formas compactas como **UTF8** u **UTF16**.

   UTF-8 permite codificar cualquiera de los más de 120.000 caracteres de Unicode y hacerlos accesibles para los ordenadores.

   En cambio, **UTF-16** utiliza 16 bits (2 bytes) para representar cada carácter Unicode, lo que significa que puede representar hasta 65.536 caracteres diferentes.

   En comparación con UTF-8, UTF-16 utiliza un número fijo de bytes para cada carácter, lo que puede hacer que los archivos de texto sean más predecibles en cuanto a su tamaño.

8. MEDICIÓN DE LA CALIDAD DE NUESTRO MODELO

La industria es sensible a cualquier fallo debido a que los errores en el reconocimiento de caracteres pueden tener consecuencias significativas, como la pérdida de datos, la interpretación incorrecta de información importante, la toma de decisiones erróneas basadas en los resultados del modelo, errores de trazabilidad, etc.

Esto equivale a que la **CER** (tasa de error de caracteres) y la **WER** (tasa de error de palabras) de nuestro modelo deben de ser nulas, lo que conlleva a que, tanto el número mínimo de inserciones de caracteres o palabras, el número mínimo de sustituciones de caracteres o palabras y el número mínimo de eliminaciones de caracteres o palabras, sean igual a cero.

Por ello, nuestro modelo ideal debería de tener una precisión lo más cercana al 100% posible, para que no haya ningún tipo de error en otros procesos de seguimiento y monitorización de la producción.

9. POSIBLES SOLUCIONES

Existen distintas soluciones en función de la superficie y la forma de impresionar los caracteres en el material.

En los casos más sencillos, ya resueltos, nos encontramos impresiones de tinta sobre papel y sucedáneos. En muchas ocasiones, con una fuente de letra fácil de interpretar. La precisión de los algoritmos actuales sobre ellos es alta, ya que la imagen tiene un gran contraste (negro sobre blanco), y la fuente está elegida.

Las primeras soluciones de Deep learning se han aplicado en este campo, con casos señalados como el dataset de MNIST (agrupación de imágenes formadas por caracteres escritos a mano), las redes convolucionales (CNN), etc..

No es un problema resuelto, aunque si avanzado.

Por último, quedan textos sobre imágenes naturales, como podrían ser cartelería, y superficies industriales. Estos suponen un reto ya que la captación de imagen no se produce en condiciones ideales. Y en el caso de las superficies industriales, la variedad de superficies y grabados es muy amplia, con graves problemas de brillos, contrastes...

Por tanto, las posibles soluciones a nuestro problema podían ser las siguientes:

1. **Visión artificial tradicional**
2. **Machine Learning**
3. **Detectores de Deep Learning**

10. ESTUDIO PREVIO

A continuación, se va a realizar un estudio de las posibles soluciones existentes para posteriormente elegir la que mejor se adapte a nuestro problema.

10.1. Visión artificial tradicional por computador

La visión artificial tradicional ha sido muy útil en el entorno industrial para una amplia variedad de tareas como la medición de dimensiones, la detección de defectos y la clasificación de objetos. Sin embargo, a medida que avanza la tecnología y al no ser muy generalista hace que prácticamente no se utilice o se utilice para tareas muy específicas y sencillas.

Las principales desventajas de esta técnica son las siguientes:

- No tiene capacidad de aprendizaje ya que depende únicamente de unas reglas preprogramadas lo que la hace poco flexible debido a que generalmente hay que adaptar cada algoritmo a cada problema.

- Tiene dificultad para adaptarse a datos complejos que se salgan del entorno de trabajo para el cual ha sido programado.

  Las fotografías analizadas por visión tradicional no pueden variar mucho sus condiciones y es conveniente que lo que se quiere detectar este suficientemente diferenciado del fondo.
  Algunos ejemplos de estas fotografías podrían ser:

Figura 15. Ejemplos de fotografía de entrada para algoritmos de visión tradicional. Fuente: https://es.123rf.com/photo_78645793_alfabeto-letras-colecci%C3%B3n-texto-letras-negras-conjunto-sobre-fondo-blanco-vector-illustration.html

El principal motivo por el cual esta tecnología es cada vez menos usada es debido a su alta sensibilidad a la varianza, lo que hace muy costoso llegar a una parametrización ideal del algoritmo o ajustar diferentes valores para llegar a tener la mayor cantidad de posibilidades en cuenta.

10.2. Machine Learning

Esta técnica ha sido y es muy útil en la solución de una amplia variedad de problemas, incluyendo la visión artificial.

Se sigue continúa utilizando cuando la separación espacial de los destinos objetos es alta. Por otro lado, permiten identificar y clasificar los caracteres, pero cuentan con pocos ajustes donde calibrar y ajustar el funcionamiento.

- Tiene una mayor precisión que la visión tradicional ya que es una técnica de aprendizaje automático que permite a los algoritmos mejorar con el tiempo, pero a un nivel inferior que el Deep Learning.

- Cuenta con una mayor flexibilidad que la visión tradicional pero menor que el Deep Learning ya que, aunque el modelo este diseñado para un problema en concreto, si durante la práctica varían en un rango no muy amplio las condiciones de trabajo, el modelo es probable que siga funcionando correctamente.

  Las fotografías analizadas por esta técnica de aprendizaje automático pueden ser más complejas en cuanto a que en la visión tradicional ya que se utilizan redes de aprendizaje para el reconocimiento de patrones.

  Un ejemplo de fotografías que se podrían analizar con esta técnica serían las que forman el dataset de **mnist**.

  Este dataset cuenta con la facilidad de tener las letras y los números muy bien diferenciados con el fondo, pero con la dificultad de ser caracteres y números escritos a mano, por lo que se convierte en un problema más complicado que ya no se podría analizar con visión tradicional.

Figura 16. Dataset de mnist. Fuente: https://learnopencv.com/tag/mnist/

Un ejemplo de modelo de Machine Learning podría ser un clasificador del dataset **mnist**.

10.3. Deep Learning

El Deep Learning es la técnica más avanzada y completa de las tres vistas.

Se ha utilizado con éxito en una amplia variedad de problemas, incluyendo la visión artificial.

En comparación con la visión artificial tradicional y el Machine Learning, el Deep Learning cuenta con varias **ventajas**.

- Ofrece la mayor precisión de los tres métodos ya que es capaz de aprender datos más complejos.

- Capacidad de aprendizaje profundo gracias a que esta técnica utiliza redes neuronales profundas para aprender, lo que permite una mayor exactitud y adaptación a los posibles cambios en el entorno de trabajo para el que haya sido diseñado.

- Mejora continua porque una vez probado el modelo se puede observar en qué casos falla y añadir esos datos/imágenes para reentrenar el modelo y que ya no se vuelva a equivocar en esos casos.

En cambio, hay algunas **desventajas** que presenta esta tecnología como:

- La necesidad de una gran cantidad de datos necesarios para su entrenamiento.

- Su consumo de recursos y su "peso" dentro del sistema en el que se ejecute.

- Es relativamente fácil que, si no se ajustan correctamente los parámetros de entrenamiento, el modelo tenga "overfitting".

Un ejemplo de imagen en la que se utilizaría esta tecnología sería:

Figura 17. Ejemplos de imágenes.

Por ejemplo, en estas imágenes lo que intentamos detectar es el número que se ve en la parte superior de la barra de metal, así como los números que se encuentran en el centro tanto a la derecha como a la izquierda.
Vemos que, aunque no estén muy bien definidos o tengan reflejos, gracias a la tecnología de Deep Learning (modelo usado: YOLO) y un buen etiquetado previo lo hemos podido detectar.

Un ejemplo de implementación de esta tecnología podría ser el modelo OCR: **EasyOCR**:

Si nos fijamos en el modelo de **EasyOCR** como ejemplo, encontramos que este algoritmo basa su modelo de reconocimiento en una *red neuronal convolucional recurrente* (CRNN) que, a su vez, está formada por un grupo de dos redes neuronales:

- *Red neuronal convolucional profunda (DCNN)*
- *Red neuronal recurrente (RNN)*

Algunas de las principales **ventajas** de las CRNN son las siguientes:

1. Puede aprender directamente de etiquetas en formato secuencia (palabras), sin necesidad de anotaciones detalladas (caracteres que formarían cada palabra).

2. Tiene la misma propiedad que las DCNN de aprender representaciones informativas directamente de los datos de la imagen, sin requerir preprocesamiento (bancarización, segmentación, localización de componentes).

3. Tiene la misma propiedad que las RNN pudiendo producir una secuencia de etiquetas.

4. No tiene limitación en cuanto a las longitudes de objetos similares a secuencias y solo requiere la acotación de la altura en las fases de entrenamiento y test.

5. Contiene menos parámetros que un modelo de red DCNN habitual y pesa menos.

Algunas de las **desventajas** que presentan estas redes son:

1. Necesidad de una cantidad adecuada de datos de entrenamiento: Al igual que con cualquier red neuronal, las CRNN requieren una cantidad suficiente de datos de entrenamiento para aprender patrones y generalizar correctamente. Si el conjunto de datos de entrenamiento es limitado, el rendimiento del modelo puede verse afectado, especialmente en tareas complejas.

2. Mayor riesgo de sobreajuste debido a su mayor capacidad y número de parámetros, estas redes son más propensas al sobreajuste en comparación con las CNN. Si no se cuenta con suficientes datos de entrenamiento, las CRNN pueden tener dificultades para generalizar correctamente con nuevos datos.

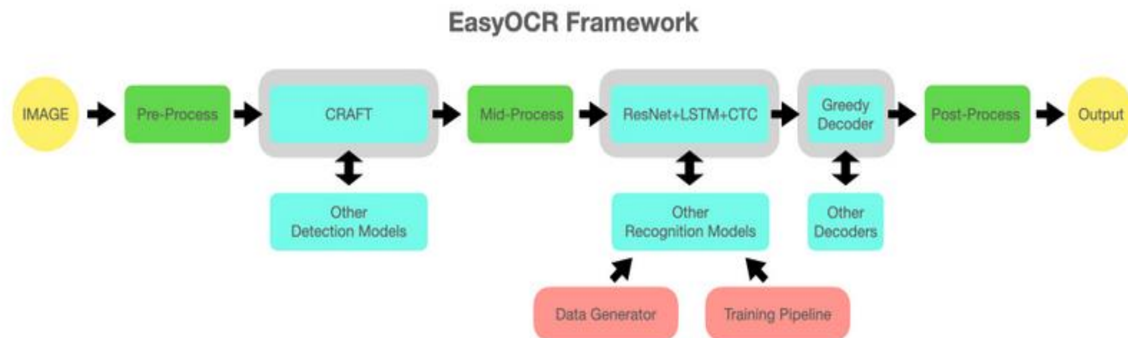


Figura 18. Flujograma de EasyOCR. Fuente: https://github.com/JaidedAI/EasyOCR

10.4. Conclusión técnica escogida

En resumen, el **Deep Learning** es una mejor opción en nuestro caso que el **Machine Learning** y la **visión tradicional** gracias a su capacidad para aprender y mejorar con el tiempo, su capacidad para manejar grandes cantidades de datos y para personalizarse y adaptarse a diferentes situaciones, es decir, ser muy robusto a la varianza.

Aunque se ha optado por utilizar **Deep Learning,** no se ha escogido el modelo **EasyOCR** ya que, al necesitar una tasa de error nula, no nos beneficios de una de las propiedades de su modelo, **la recursión**, debido a que necesitamos que detectar una palabra con 0 fallos en contra de elaborar textos que se ajusten con diccionarios.

De esta manera, elegiremos el Deep Learning ya que vamos a tener escenarios de trabajo muy distintos unos de otros con distintas condiciones y ahí es donde la tecnología de Deep Learning destaca.

11. DESARROLLO DEL PROGRAMA

En este apartado se va a explicar las distintas fases del apartado de desarrollo, tanto los scripts que intervienen en la preparación de los datos como el posterior entrenamiento del modelo.

El flujo de proceso a seguir será el siguiente:

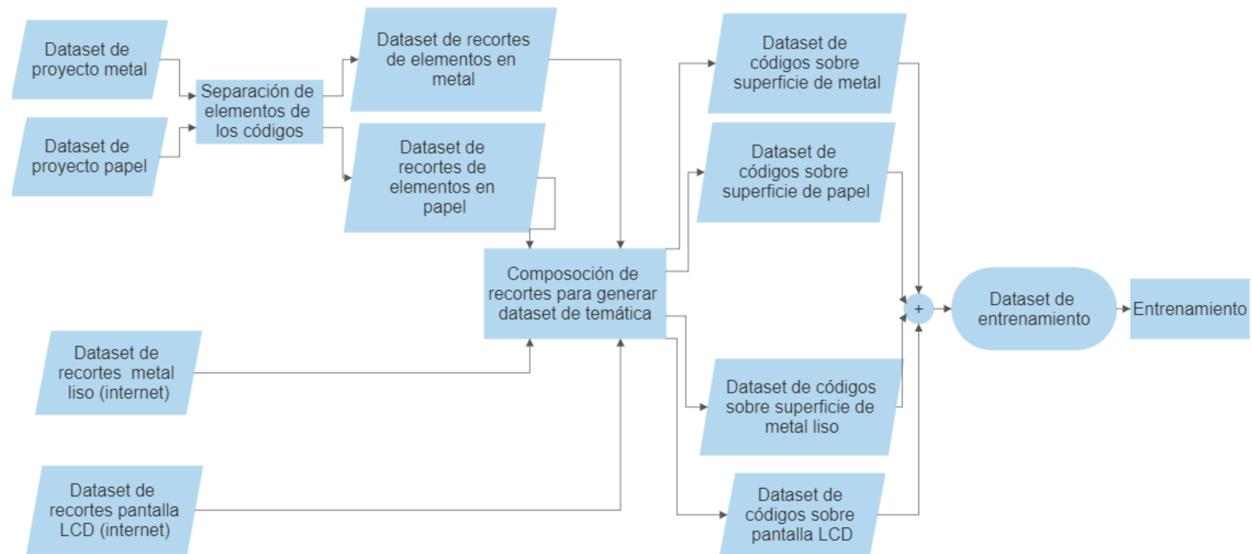


Figura 19: Flujograma de proceso. Fuente: Elaboración propia.

El primer motivo por el cual hemos decidido hacer un **generador de datos** es debido a que un modelo de Deep Learning necesita ser alimentado con una gran cantidad de datos dotados de mucha varianza, pero manteniendo la **calidad** de estos.

Con la **calidad** de los datos nos referimos a la precisión, integridad, consistencia y relevancia de los datos utilizados para entrenar el modelo, es importante que los datos sean representativos y estén bien etiquetados para que el modelo pueda aprender de manera efectiva.

Por ejemplo, si los textos a detectar siempre son negros sobre fondo blanco, los datos de entrenamiento con variaciones de color en el texto o el fondo aportarán poco valor, mientras que variaciones en el tamaño de letra pueden ser muy útiles si la cámara puede estar a distintas distancias.

Es muy relevante la calidad de un dataset ya que, en nuestro caso, no disponemos de diccionarios de comparación con los que podamos corregir los posibles errores debido a que la variedad de palabras en los códigos de trazabilidad suele ser muy amplia, no limitada a miles como en un lenguaje, no pudiendo corregir errores de carácter.

El segundo motivo es la escasez de este tipo de datasets en internet (lo que limita la innovación en esta tecnología debido al elevado coste en tiempo de elaborar un buen dataset) y lo que nos dio la idea de crear un dataset lo más completo posible a partir de otros.

Pero la generación de imágenes de manera artificial presenta un inconveniente, la pureza de una foto real no es la misma que la de una foto generada, es decir, la cantidad frente a la calidad, algo relativamente aceptado en una tecnología basada en la aproximación por repetición.

Algunas varianzas que se pueden aplicar a una imagen son:

- Rotación
- Adición de ruido
- Variaciones de exposición
- Desenfoque de movimiento, etc...

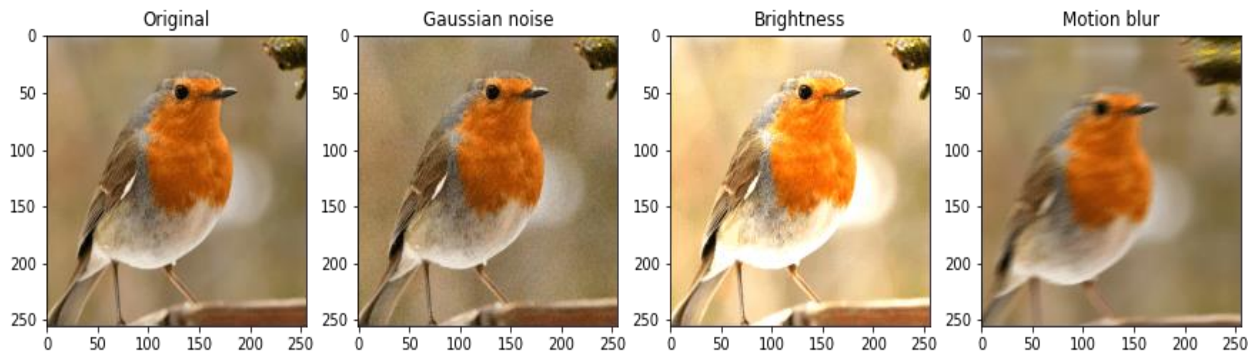


Figura 20. Ejemplos de varianzas. Fuente: https://dev.to/itminds/improve-your-deep-learning-models-with-image-augmentation-4j7f

Por tanto, se ha ideado el siguiente generador de datos para tratar de tener el número máximo de datos posibles con fin de que el modelo sea lo más preciso posible.

Este generador recoge imágenes de caracteres en distintas superficies (metal, plástico, madera) y acabados. Siguiendo una distribución de números y letras indicada, recombina diferentes imágenes de caracteres de forma aleatoria. De esta forma, obtenemos un gran número de imágenes con códigos diferentes, con diferentes condiciones y sobre la superficie deseada.

A continuación, se explicará detalladamente la solución que hemos decidido implementar para solventar el problema mencionado anteriormente.

11.1. Entradas y salidas

- Entradas:
  - Número de imágenes que queremos generar
  - Cantidad máxima de letras y de números que formaran el código de la imagen final.
  - Ruta al dataset con ejemplos de números y letras (podemos crearlo nosotros mismos con un script que se explicará posteriormente).
  - Ruta al archivo de las clases que vamos a tener
  - Ruta al fondo que queremos en las imágenes finales.
  - Offset: distancia entre las letras del código final.

- Salidas:
    - Carpeta “images” que contendrá las imágenes en formato .jpg con los códigos generados.

      La composición de estas imágenes se realizará mediante una técnica llamada **tiling**, una técnica de diseño que se basa en utilizar patrones de imágenes diferentes en forma de baldosas (rectangulares) un ejemplo similar podría ser un mosaico.

      Algunos ejemplos de estas imágenes podrían ser:

Figura 21. Ejemplos de imágenes de códigos generadas con nuestro programa. Fuente: Elaboración propia

11.2. Estructura

El generador de datos va a contar con varios archivos para su correcto funcionamiento, los archivos y carpetas que formarán este generador serán los siguientes:

- Carpeta **cfgs**: en esta carpeta se almacenarán los datos que el usuario introduce mediante la API en un archivo .json, en nuestro caso esta carpeta contendrá dos archivos.

    - **Parameters_dataset_input.json**: conjunto de parámetros que se aplicarán para la generación del dataset de entrada (recortes de números y letras).

```
{} parameters_dataset_input.json ×
cfgs > {} parameters_dataset_input.json > ...
1  {
2      "font": 1,
3      "path_background": "fondo1.jpg",
4      "offset": 5
5  }
```

Figura 22. Parámetros para generar recortes. Fuente: Elaboración propia

Font: fuente correspondiente al número asignado, con esta variable se podrá decidir que fuente queremos en los recortes de las imágenes (número del 0 al 7).

Path_background: fondo escogido por el usuario, esta variable nos permitirá generar recortes de letras y números con el fondo que nosotros queramos, aportando mucha variabilidad al dataset de entrada.

Offset: “marcos” que tendrán los recortes de las imágenes, utilizado para otorgar variabilidad en cuanto al tamaño del elemento.

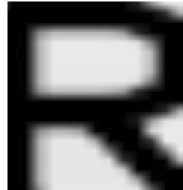


Figura 23. Offset bajo. Fuente: Elaboración propia

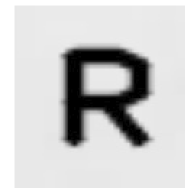


Figura 24. Offset alto. Fuente: Elaboración propia

- **Parameters_dataset_output**: parámetros generales para la composición de imágenes que formaran el dataset final con el que se entrenará el modelo.

```
{} parameters_dataset_output.json ×
cfgs > {} parameters_dataset_output.json > ...
1  {
2      "num_img": 5,
3      "num_letters": 5,
4      "num_numbers": 5,
5      "path_dataset": "/src_dataset/dataset",
6      "path_classes": "/src_dataset/classes.txt",
7      "path_background": "fondo1.jpg",
8      "offset": -4
9  }
```

Figura 25. Parámetros para la realización de códigos. Fuente: Elaboración propia

- Num_img: este número corresponde con el número final de imágenes que vamos a querer en nuestro dataset, el paso previo al entrenamiento del modelo.
- Num_letters: número máximo de letras que se van a generar por código.
- Num_numbers: cantidad máxima de números que se van a generar por código.
- Path_dataset: ruta al dataset que generaremos con el script generar_dataset, este conjunto de datos se utilizara como entrada en el programa principal
- Path_classes: ruta al archivo “classes.txt” que contendrá todas las clases posibles que se pueden dar.
- Path_background: ruta al fondo para la composición de imágenes, es altamente recomendable elegir el mismo fondo que en la generación del dataset de entrada
- Offset: separación entre los recortes de los números o letras al formar el código final

- Carpeta **results**: carpeta que se creará cuando se ejecute el programa, dentro de esta carpeta tendremos ostras dos:
    - Carpeta “images”: aquí se almacenarán los códigos creados con el programa.
    - Carpeta “labels”: aquí se almacenarán las etiquetas de los códigos de cada imagen generada
    - Diccionario_clases.json: archivo que nos permitirá tener información sobre los códigos generados, diciéndonos cuantas veces se repite cada elemento.


```
{"0": 1, "1": 0, "2": 1, "3": 0, "4": 0, "5": 2, "6": 0, "7": 0, "8": 2, "9": 1, "A": 0, "B": 0, "C": 0, "D": 0, "E": 0, "F": 0, "G": 0, "H": 0, "I": 0, "J": 0, "K": 0, "L": 0, "M": 0, "N": 0, "O": 0, "P": 0, "Q": 1, "R": 0, "S": 0, "T": 0, "U": 0, "V": 0, "W": 0, "X": 0, "Y": 0, "Z": 0, "a": 0, "b": 0, "c": 0, "d": 0, "e": 0, "f": 0, "g": 0, "h": 0, "i": 0, "j": 0, "k": 0, "l": 1, "m": 0, "n": 0, "o": 0, "p": 0, "q": 1, "r": 0, "s": 0, "t": 0, "u": 1, "v": 1, "w": 0, "x": 0, "y": 0, "z": 0}
```


Figura 26. Diccionario de registro de elementos generados en los códigos. Fuente: Elaboración propia

- **Docker-compose.yml**: archivo con el servicio de Docker para la API que se conectará con el puerto 5000 del ordenador en el que se ejecute.

```
version: '2.3'
services:
  api:
    build: .
    ports:
    - 5000:5000
    volumes:
      - "./src_api:/src_api/"
      - "./src_dataset:/src_dataset/"
      - "./cfgs/:/cfgs/"
      - "./results/:/results/"
    stdin_open: true
    command: ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "5000", "--debug"]
```

Figura 27. Archivo de Docker: docker-compose.yml. Fuente: Elaboración propia

Este generador de datos sigue la filosofía de contenedores (Dockers).

Los contenedores (Dockers) son un método de virtualización ligero que aísla una aplicación y sus dependencias en un paquete de software portátil, lo que permite ejecutar la aplicación en cualquier entorno que admita contenedores.

Ese archivo siempre va acompañado del archivo “Dockerfile” (explicado posteriormente) el cual contendrá las propiedades del servicio

- **Dockerfile**: archivo necesario para la creación del Docker donde se especificarán características del contenedor (versión de Python, directorio de trabajo, requerimientos, etc.…).

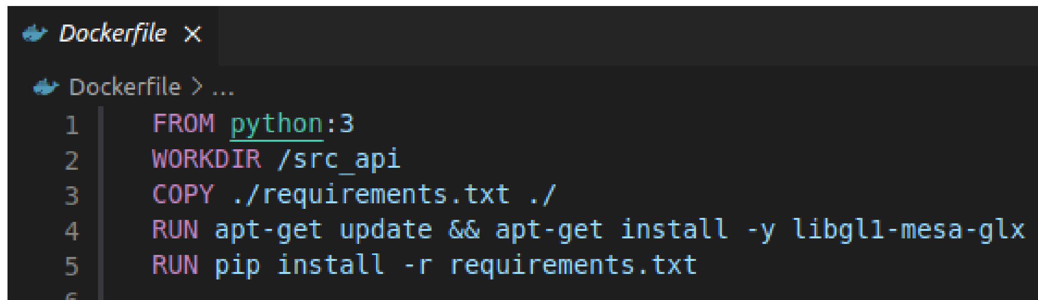


Figura 28. Archivo Dockerfile. Fuente: Elaboración propia

- **Requierements.txt**: documento de texto con las librerías necesarias para el correcto funcionamiento del Docker, se instalará en el contenedor gracias que se correrá en el archivo **Dockerfile** mediante el atributo "—build" al levantar el archivo **docker-compose**.

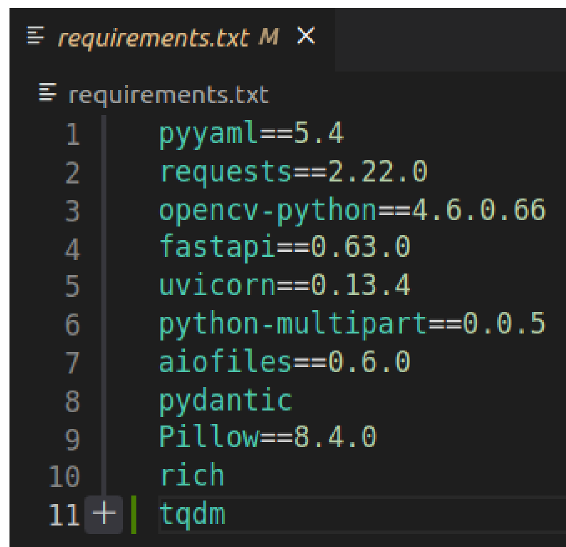


Figura 29. Archivo requirements.txt. Fuente: Elaboración propia

- **Readme.md**: Documento donde se explicará el funcionamiento del programa

- Carpeta **src_api**: en esta carpeta se encontrará el corazón del programa, aquí estarán todos los scripts necesarios, así como el resto de las carpetas necesarias para el funcionamiento del programa.
  - **Fastapi_models**.py: script que constara de una clase:

```python
from enum import Enum
from os import getenv

class ApiResponse:

    def __init__(self, success=True, data=None, error=None):
        """
        Defines the response shape.

        Args:
            success (bool): A boolean that returns if the request has \
                succeeded or not.
            data (JSON-object): The model's response.
            error (JSON-object): The error in case an exception was raised.
        """
        self.success = success
        self.error = error
        self.data = data
```

Figura 30. Script de configuración de las respuestas de la API. Fuente: Elaboración propia

    - ApiResponse: posibles respuestas para la API

  - **Main.py:** este script se encargará de manejar la API mediante dos métodos:

    - **Dataset_image_parameters**: este método lo utilizaremos para la generación del dataset formado por recortes de números y letras.

```python
@app.post('/dataset_image_parameters')
async def dataset_image_parameters_execute(
        font: int = 1,
        path_background : str =  Query(None, enum=os.listdir('/src_dataset/fondos/')),
        offset: int = 5):
    parameters = {'font': font,
                  'path_background': path_background,
                  'offset': offset}
    if not os.path.exists(PARAMETERS_FOLDER):
        os.makedirs(PARAMETERS_FOLDER)
    filename = "parameters_dataset_input.json"
    with open(os.path.join(PARAMETERS_FOLDER, filename), 'w') as file:
        json.dump(parameters, file)

    subprocess.run(["python", "/src_dataset/generar_dataset.py"])
    return ApiResponse()
```

Figura 31. Método para la generación del dataset de recortes de números y letras. Fuente: Elaboración propia

Como parámetros de entrada tendremos.

1. Fuente escogida.
2. Desplegable con las rutas a los fondos disponibles
3. Offset: número correspondiente a los marcos que tendrá la imagen de cada recorte.

Se comprobará la existencia de la carpeta “cfgs”.

Se guardarán todos los parámetros en un archivo “.json”

Se llama al programa “generar_dataset.py” encargado de generar el dataset de entrada para el siguiente método.

Como salida tendremos la respuesta de la API.

- **Previous_parameters:** método que se encargara de coger los parámetros que el usuario introduzca mediante la API y ejecutar el programa con dichos parámetros.

```
@app.post('/previous_parameters')
async def load_parameters_and_execute(
        num_img: int ,
        num_letters : int,
        num_numbers : int,
        path_dataset: str = '/src_dataset/dataset',
        path_classes : str = '/src_dataset/classes.txt',
        path_background : str =  Query(None, enum=os.listdir('/src_dataset/fondos/')),
        offset: int = -4):
    parameters = {'num_img': num_img,
                  'num_letters': num_letters,
                  'num_numbers': num_numbers,
                  'path_dataset': path_dataset,
                  'path_classes': path_classes,
                  'path_background': path_background,
                  'offset': offset}
    if not os.path.exists(PARAMETERS_FOLDER):
        os.makedirs(PARAMETERS_FOLDER)
    filename = "parameters.json"
    with open(os.path.join(PARAMETERS_FOLDER, filename), 'w') as file:
        json.dump(parameters, file)

    subprocess.run(["python", "/src_dataset/main_dataset.py"])
    return ApiResponse()
```

Figura 32. Método utilizado para la generación del dataset de códigos. Fuente: Elaboración propia

Como parámetros de entrada tendremos.

1. Numero de imágenes.
2. Cantidad máxima de letras que formaran el código.
3. Cantidad máxima de números que formaran el código.
4. Ruta al dataset generado en el apartado anterior.
5. Ruta al archivo de las clases que tendremos.
6. Menú desplegable con las imágenes de fondo disponibles.
7. Offset para elegir la separación entre letras.

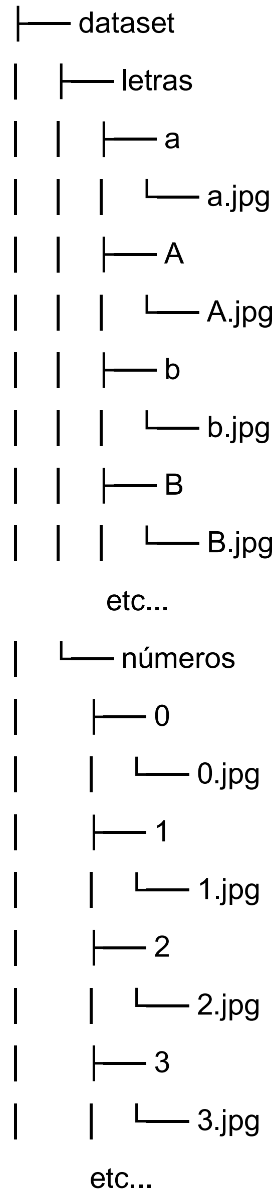
Se comprobará la existencia de la carpeta “cfgs”.

Se guardarán todos los parámetros en un archivo “.json”

Se llama al programa “main_datasetpy” responsable de la composición de imágenes.

Como salida tendremos la respuesta de la API.

- **Carpeta “src_dataset”:** en esta carpeta se van a encontrar todos los scripts de tratamiento de imágenes y generación del código.
  - Carpeta “dataset”: en esta carpeta se encuentran las imágenes clasificadas tanto de números como de letras. Siguiendo la siguiente estructura:



Este conjunto de imágenes servirá como argumentos de entrada para el programa de tratamiento de imágenes.

Cabe destacar que este dataset lo podremos generar nosotros haciendo uso del script **generar_dataset** explicado a continuación.

  - Carpeta **fondos**: en esta carpeta se encontrarán todas las imágenes que queramos utilizar como fondo en la generación de códigos en formato .jpg.

- Archivo **classes.txt:** archivo con todas las clases posibles, tanto números como letras. En este caso letras de la “A” a la “Z” tanto en mayúsculas como en minúsculas, así como números del 0 al 9.

  Este archivo servirá para la creación del diccionario de las clases con objetivo de ver la calidad del dataste de salida.

- Archivo **classes_let.txt:** archivo con las letras que queramos en los códigos del dataset final.
- Archivo **classes_num.txt:** archivo con los números que queramos en los códigos del dataset final.
- Script **generar_dataset**: Este script lo utilizaremos para generar nosotros mismo los recortes de los números y de las letras que queramos. Es recomendable utilizarlo cuando el objetivo de detección del modelo de OCR no sea muy complicado, es decir, generalmente números o letras escritas por ordenador.

  La forma de interactuar con este script será mediante la API, podremos elegir el número de fuentes que queremos, el fondo que deseamos para la generación de imágenes de recortes, y el offset (marcos de los recortes).

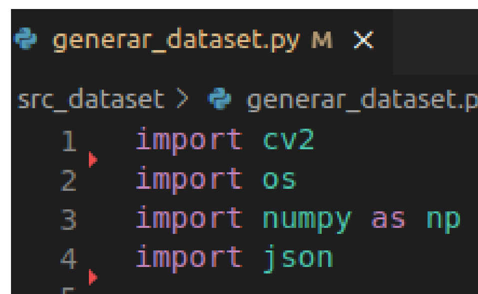


Figura 33. Librerías utilizadas. Fuente: Elaboración propia

Este script cuenta con varias funciones

- Función **main:** función principal del programa

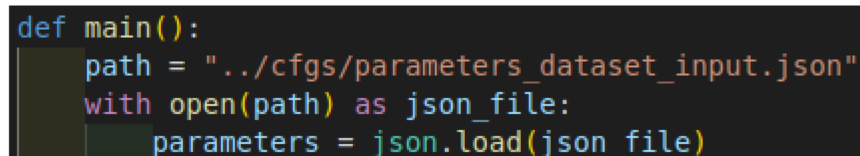


Figura 34. Inicio función principal. Fuente: Elaboración propia

Cargamos los parámetros necesarios dados por el usuario mediante la API

Apertura de archivos de clases:

```python
with open("../src_dataset/classes_num.txt", "r") as f1:
    lista_nums = []
    for linea in f1:
        lista_nums.append(linea.strip())

with open("../src_dataset/classes_let.txt", "r") as f2:
    alphabet = []
    for linea in f2:
        alphabet.append(linea.strip())
```

Figura 35. Generación de listas. Fuente: Elaboración propia

Generamos la lista de números y la lista de letras haciendo uso de los archivos clsses_numeros.txt y classes_letras.txt.

Comprobación de directorios:

```python
path_background = os.path.join("../src_dataset/fondos",
                               parameters["path_background"])
fondo = cv2.imread(path_background)
alto = fondo.shape[0]
ancho = fondo.shape[1]
xy = (int(ancho/2), int(alto/2))
if not os.path.exists('../src_dataset/dataset'):
    os.makedirs('../src_dataset/dataset')
if not os.path.exists('../src_dataset/dataset/letras'):
    os.makedirs('../src_dataset/dataset/letras')
if not os.path.exists('../src_dataset/dataset/numeros'):
    os.makedirs('../src_dataset/dataset/numeros')
```

Figura 36. Configuración de carpetas. Fuente: Elaboración propia

Establecemos las coordenadas donde se escribirá el texto y comprobamos la existencia de las carpetas necesarias.

Llamado a funciones:

```python
    if not os.path.exists('../src_dataset/dataset/numeros'):
        os.makedirs('../src_dataset/dataset/numeros')

    generar_letras(fondo, alphabet, parameters["font"], parameters["offset"], xy)
    generar_numeros(fondo, lista_nums, parameters["font"], parameters["offset"], xy)

if __name__ == '__main__':
    main()
```

Figura 37. Funciones de tratamiento de datos. Fuente: Elaboración propia

Llamamos a las dos funciones de tratamiento de imágenes para la generación de recortes y finalmente ejecutamos la función **main.**

- Función **generar_letras:** función encargada de generar los recortes de las letras que nosotros establezcamos en el archivo classes_letras.txt.

```
def generar_letras(fondo, abecedario , fuente, offset, xy):
    """Funcion que nos va a generar recortes de imagenes
        de letras en mayusculas y minusculas

    Args:
        fondo: (imagen): imagen de fondo
        abecedario (lista): Lista con todas las letras del
            abecedario en mayusculas
        fuente (int): fuente que se va a utilizar para la
            generacion de imagenes
        offset (int): variacion en los margenes del recorte
            de la imagen
        xy: coordenadas para la posicion del texto
    """
```

Figura 38. Breve descripción de la función. Fuente: Elaboración propia

Bucle principal de esta función:

```
for letra in abecedario:
    fondo_copy1 = fondo.copy()
    cv2.putText(fondo_copy1, letra, xy, fuente, 2,
                (0, 0, 0), 2, cv2.LINE_AA)
    fondo_copy2 = fondo_copy1.copy()
    fondo_gris = cv2.cvtColor(fondo_copy2, cv2.COLOR_BGR2GRAY)
    th = cv2.threshold(fondo_gris, 100, 255, cv2.THRESH_BINARY_INV)[1]
    contours = cv2.findContours(th, 1, 2)[0]
    contours = sorted(contours, key=cv2.contourArea, reverse=True)
    x = []
    y = []
```

Figura 39. Bucle principal de la función. Fuente: Elaboración propia

Iteraremos tantas veces como letras haya en el archivo classes_letras.txt.

En cada iteración se realizará la escritura de una letra en la foto de fondo que se ha elegido, y, gracias a la umbralización y a la detección de contornos de la librería **opencv** obtendremos los contornos de mayor área de la letra que se haya escrito.

Hay dos letras sensibles a la detección de contornos, ya que cuentan con dos contornos de diferente tamaño (la i y la j). Para solventar este problema hay que establecer una condición cuando se lleguen a dichas letras.

Condición de letras especiales (i, j):

```
x = []
y = []
if ord(letra) == 105 or ord(letra) ==106:
    pass
    contorno_mayor = contours[0]
    contorno_2mayor = contours[1]
    contorno_letra = np.concatenate((contorno_mayor,
                                     contorno_2mayor),
                                     axis=0)
    for lista in contorno_letra:
        x.append(lista[0][0])
        y.append(lista[0][1])
    max_x = max(x)
    min_x = min(x)
    max_y = max(y)
    min_y = min(y)
else:
    contorno_letra = contours[0]
    for lista in contorno_letra:
        x.append(lista[0][0])
        y.append(lista[0][1])
    max_x = max(x)
    min_x = min(x)
    max_y = max(y)
    min_y = min(y)
```

Figura 40. Condición letra especial. Fuente: Elaboración propia

En el momento en el que la letra sea "i" o "j" entrará la condición, lo que se hará será concatenar los dos mayores contornos de la letra (el punto y la letra en sí).

Posteriormente, los puntos "x" e "y" de los contornos se agregan a las listas "x" e "y" respectivamente.

Finalmente, se obtienen los valores máximos y mínimos de "x" e "y" en el contorno de la letra.

En caso de que la condición no se cumpla (resto de letras), se hará lo mismo, pero únicamente fijándonos en el contorno de mayor área.

Recortado y guardado de la imagen.

```
img_recortada = fondo_copy1[min_y - offset:max_y + offset, min_x - offset:max_x + offset]
img_recortada = cv2.resize(img_recortada, (128, 128))
if not os.path.exists(f'../src_dataset/dataset/letras/{letra}'):
    os.makedirs(f'../src_dataset/dataset/letras/{letra}')
cv2.imwrite(f'../src_dataset/dataset/letras/{letra}/{letra}.jpg',img_recortada)
```

Figura 41.

Para finalizar la función, se recortará la imagen de la letra haciendo uso de las coordenadas máximas y mínimas de "x" e "y" jugando con el offset para la definición de los marcos y se guardará la imagen en la carpeta correspondiente a la letra que se haya generado.

Figura 42. Offset = 1. Fuente: Elaboración propia

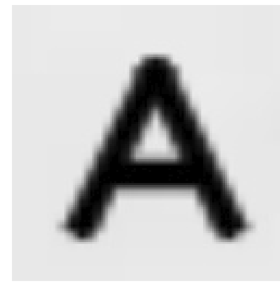


Figura 43. Offset = 6. Fuente: Elaboración propia

- Función **generar_numeros:** esta función será muy similar a la anterior a excepción de la condición de la "i" y la "j".

```
def generar_numeros(fondo, lista_numeros, fuente, offset, xy):
    """Funcion que va a generar recortes de imagenes del 0 al 9
    Args:
        fondo: (imagen): imagen de fondo
        lista_numeros (lista): lista con los numeros del 0 al 9
        fuente (int): fuente que se va a utilizar para la
            generacion de imagenes
        offset (int): variacion en los margenes del recorte
            de la imagen
        xy: coordenadas para la posicion del texto
    """
```

Figura 44. Breve descripción de la función. Fuente: Elaboración propia

A continuación, se procederá con la explicación de la generación de números:

```python
for _, numero in enumerate(lista_numeros):
    fondo_copy3 = fondo.copy()
    cv2.putText(fondo_copy3, str(numero), xy, fuente, 2, (0, 0, 0), 2, cv2.LINE_AA)
    fondo_copy4 = fondo_copy3.copy()
    fondo_gris = cv2.cvtColor(fondo_copy4, cv2.COLOR_BGR2GRAY)
    th = cv2.threshold(fondo_gris, 100, 255, cv2.THRESH_BINARY_INV)[1]
    contours = cv2.findContours(th, 1, 2)[0]
    contours = sorted(contours, key=cv2.contourArea, reverse=True)
    contour = contours[0]
```

```python
    x = []
    y = []
    for lista in contour:
        x.append(lista[0][0])
        y.append(lista[0][1])
    max_x = max(x)
    min_x = min(x)
    max_y = max(y)
    min_y = min(y)
    img_recortada = fondo_copy3[min_y - offset:max_y + offset, min_x - offset:max_x + offset]
    img_recortada = cv2.resize(img_recortada, (128, 128))
    if not os.path.exists(f'../src_dataset/dataset/numeros/{numero}'):
        os.makedirs(f'../src_dataset/dataset/numeros/{numero}')
    cv2.imwrite(f"/src_dataset/dataset/numeros/{numero}/{numero}.jpg",img_recortada)
```

Figura 45. Función utilizada para la generación de números. Fuente: Elaboración propia

Al igual que en la función anterior, realizamos la umbralización del número que se va a escribir, detectamos contornos y nos quedamos con el mayor de ellos.

Posteriormente, los puntos “x” e “y” de los contornos se agregan a las listas “x” e “y” respectivamente.

Finalmente, se obtienen los valores máximos y mínimos de “x” e “y” en el contorno del número, se recortará la imagen del número haciendo uso de las coordenadas máximas y mínimas de “x” e “y” usando el offset para la definición de los marcos y se guardará la imagen en la carpeta correspondiente al número que se haya generado.

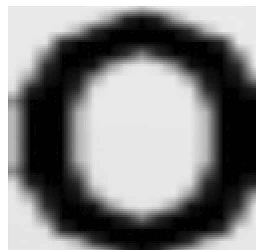


Figura 46. Offset = 0. Fuente: Elaboración propia

Figura 47. Offset = 6. Fuente: Elaboración propia

- Script **logger_handler**: este script nos permitirá que nuestro debuggin sea más claro gracias a la estructura de impresión por pantalla.

```python
import logging
from rich.logging import RichHandler
from datetime import datetime

logger = logging.getLogger(__name__)

shell_handler = RichHandler(markup=True,
                            omit_repeated_times=False,
                            log_time_format="[%d/%m/%y %H:%M:%S]")
timestamp = datetime.now().strftime('%Y_%m_%d_%H_%M_%S')
# file_handler = logging.FileHandler("./logs/{}_pyplc.log".format(timestamp))

logger.setLevel('DEBUG')
shell_handler.setLevel('DEBUG')
# file_handler.setLevel('DEBUG')

fmt_shell = '%(message)s'
fmt_file = '%(levelname)s %(asctime)s \
        [%(filename)s:%(funcName)s:%(lineno)d] %(message)s'

shell_formatter = logging.Formatter(fmt_shell)
file_formatter = logging.Formatter(fmt_file)

shell_handler.setFormatter(shell_formatter)
# file_handler.setFormatter(file_formatter)

logger.addHandler(shell_handler)
# logger.addHandler(file_handler)
logger.propagate = False
```

Figura 48. Script para la generación de logs. Fuente: Elaboración propia

- Script **main_dataset**: este script es el corazón de la generación de códigos ya que va a manejar las dos funciones principales.

```python
import os
import utils_dataset
import json
from rich.pretty import pprint
from logger_handler import logger


def main():
    path = "../cfgs/parameters_dataset_output.json"
    with open(path) as json_file:
        parameters = json.load(json_file)
    logger.debug("JSON loaded")
    logger.debug(parameters)
```

Figura 49. Inicio de script principal. Fuente: Elaboración propia

Importación de librerías, definición de la función principal.

Cargamos el archivo de los parámetros guardados en la carpeta “cfgs”, estos parámetros serán introducidos por el usuario mediante la API

Comprobamos existencia de directorios:

```
17
18      os.makedirs('../results/images', exist_ok = True)
19      os.makedirs('../results/labels', exist_ok = True)
20
```

Figura 50: Comprobación de directorios. Fuente: Elaboración propia

Comprobamos existencia de las carpetas que se muestran y si no existen se crearan.

Generación de bucle para llamado a funciones de tratamiento de imágenes:

```
path_background = os.path.join("../src_dataset/fondos", parameters["path_background"])
diccionario_clases = utils_dataset.generar_diccionario()
for cnt in range(parameters['num_img']):
    lista_img_codigo, dict_clases = utils_dataset.generar_codigo(parameters['path_dataset'],
                                                                 parameters['num_letters'],
                                                                 parameters['num_numbers'],
                                                                 diccionario_clases)
    utils_dataset.pegar_codigo(lista_img_codigo, path_background,
                               parameters['path_classes'],
                               dict_clases, parameters['offset'],
                               cnt)
```

Figura 51. Bucle para tratamiento de imágenes. Fuente: Elaboración propia

Generamos un diccionario de clases para tener posteriormente una retroalimentación de la cantidad de números y letras que se han generado.

Iniciamos un bucle que itere tantas veces como imágenes se quieran generar.

Dentro del bucle llamará a la función **generar_código**, cuyo funcionamiento a grandes rasgos es utilizar las imágenes al azar del dataset de entrada y juntarlas creado un código de números y letras para luego pegarlas en un fondo.

También se llamará a la función **pegar_codigo** cuyo funcionamiento a grandes rasgos es coger el código generado por la función anterior, leer las imágenes que se especifiquen en el código, y montarlas en una imagen siguiendo la técnica **tiling** pegándolas sobre un fondo.

Por último, establecemos la ruta de guardado del diccionario con los resultados y llamamos a la función main.

```
31      with open("../results/diccionario_clases.json", "w") as file:
32          json.dump(dict_clases, file)
33      logger.info("\nFINAL DEL CODIGO\n")
34
35
36  if __name__ == '__main__':
37      main()
```

Figura 52. Generación y guardado del archivo .json de registro. Fuente: Elaboración propia

- Script **utills_dataset**: en este script se van a encontrar las dos funciones principales para la generación de imágenes.
    - Función **generar_codigo**: como su propio nombre indica, es una función que nos va a generar el código.

      Esto se llevará a cabo cogiendo las rutas de las imágenes del dataset de entrada para luego ju8ntarlas en una lista y tener una lista de los números y las letras que formaran el código.

```
utils_code.py 2 X
src_dataset > utils_code.py > ...
1  import os
2  import random
3  from PIL import Image
4  from os import getenv
5  from logger_handler import logger
6  import string
7  from datetime import datetime
8
```

Figura 53. Importación de librerías necesarias. Fuente: Elaboración propia

Función encargada de la generación del código:

```
def generar_codigo(path, num_letras, num_numeros, diccionario):
    """Funcion que nos va a generar un codigo de letras y numeros
        al azar
    Args:
        path (string): ruta a la carpeta donde dentro deberar
            tener otras dos separando carpetas de numeros y letras
        num_letras (int): numero de letras que se van a querer
            en el codigo
        num_numeros (int): numero de numeros que se van a querer
            en el codigo
        dicionario: diccionario con el regustro de los numeros /
            letras que se generan
    Returns:
        lista: lista con el nombre de las imagenes de las fotos
            que van a formar parte del codigo
    """
```

Figura 54. Breve descripción de la función. Fuente: Elaboración propia

Declaración de variables necesarias;

```
rand_nums = random.randint(1, num_numeros)
rand_letras = random.randint(1, num_letras)
lista_img_numeros = []
lista_img_letras = []
lista_numeros = []
lista_letras = []
```

Figura 55. Declaración de variables necesarias. Fuente: Elaboración propia

- Rand_nums: se seleccionará un numero aleatorio entre 1 y la cantidad de números introducidos por el usuario mediante la API.
- Rand_letras: se seleccionará un numero aleatorio entre 1 y la cantidad de letras introducidas por el usuario mediante la api.
- Lista_img_numeros: lista formada por las rutas de los números que formaran el código (se rellenará en el bucle que a continuación se explicará)
- Lista_img_letras: lista formada por las rutas de las letras que formaran el código (se rellenará en el bucle que a continuación se explicará)
- Lista_numeros: lista formada por los números que formaran la imagen en formato texto (1,2.3...)
- Lista_letras: lista formada por las letras que formaran la imagen en formato texto (A, b, c, D...)

Bucle destinado a la selección aleatoria de números que formaran el código:

```
for cnt_num in range(rand_nums):
    with open("../src_dataset/classes_num.txt", "r") as f:
        lineas_num = []
        for linea in f:
            lineas_num.append(linea.strip())
    rand1 = random.randint(0,len(lineas_num)-1)
    listado_carpetas_num = os.listdir(f"{path}/numeros/{lineas_num[rand1]}")
    diccionario[str(rand1)] += 1
    lista_img_numeros.append(f"{path}/numeros/{str(rand1)}/{random.choice(listado_carpetas_num)}")
    numero = lista_img_numeros[cnt_num].split("/")[-2]
    lista_numeros.append(numero)
```

Figura 56. Bucle para la selección de números que formarán el código. Fuente: Elaboración propia

- Iteraremos tantas veces como cantidad de números hayamos definido previamente.
- Cargamos el archivo “classes_num.txt” que contendrá los números elegidos para la generación de códigos.
- Establecemos un numero aleatorio del 0 a la longitud de la lista de números (-1 ya que la función “len” empieza a contar desde 1) para poder seleccionar alguna de las carpetas de números que tengamos en nuestro dataset (0, … ,9).
- Listado_imgs_num: listado de imágenes de un determinado número.
- Diccionario para el registro de la generación de números, es decir, con este diccionario sabremos que números tenemos en los de las imágenes generadas.
- Lista_img_numeros: listado aleatorio de la ruta a las imágenes de los números que formaran el código. Esta lista se utilizará posteriormente para abrir la ruta las imágenes.
- Número: Dividiremos la cadena de cada posición de la lista_img_numeros para poder mostrar el número que se ha seleccionado posteriormente.
- Lista_numeros: lista formada por los números que formaran el código.

El siguiente bucle será muy parecido al anterior, pero en este caso con letras.

```
for cnt_let in range(rand_letras):
    with open("../src_dataset/classes_let.txt", "r") as f:
        lineas_let = []
        for linea in f:
            lineas_let.append(linea.strip())
    rand2 = random.randint(0,len(lineas_let)-1)
    listado_carpetas_letras = os.listdir(f"{path}/letras/{lineas_let[rand2]}")
    diccionario[lineas_let[rand2]] += 1
    lista_img_letras.append(f"{path}/letras/{lineas_let[rand2]}/{random.choice(listado_carpetas_letras)}")
    letra = lista_img_letras[cnt_let].split("/")[-2]
    lista_letras.append(letra)
```

Figura 57 .Bucle para la selección de letras que formarán el código. Fuente: Elaboración propia

Al igual que en bucle anterior, iteraremos tantas veces como numero de letras hayamos obtenido aleatoriamente.

- Iteraremos tantas veces como cantidad de letras hayamos definido previamente.
- Cargamos el archivo “classes_let.txt” que contendrá las letras elegidas para la generación de códigos.
- Establecemos un numero aleatorio del 0 a la longitud de la lista de números (-1 ya que la función “len” empieza a contar desde 1) para poder seleccionar alguna de las carpetas de letras que tengamos en nuestro dataset (A, a… ,Z, z).
- Listado_imgs_letras: listado de imágenes de una determinada letra.
- Diccionario para el registro de la generación de números, es decir, con este diccionario sabremos que números tenemos en los de las imágenes generadas.
- Lista_img_letras: listado aleatorio de la ruta a las imágenes de las letras que formaran el código. Esta lista se utilizará posteriormente para abrir la ruta las imágenes.
- Letra: Dividiremos la cadena de la cada posición de la lista_img_letras para poder mostrar la letra que se ha seleccionado posteriormente.
- Lista_letras: lista formada por las letras que formaran el código.

Logs y mezclado de los códigos de letras y de números.

```
logger.debug(f"Lista de numeros: {lista_numeros}")
logger.debug(f"Longitud lista numeros: {len(lista_img_numeros)}")

logger.debug(f"Lista de letras: {lista_letras}")
logger.debug(f"Longitud lista letras: {len(lista_img_letras)}")
lista_imgs_codigo = lista_img_letras + lista_img_numeros
lista_codigo = lista_letras + lista_numeros
random.shuffle(lista_imgs_codigo)
random.shuffle(lista_codigo)

logger.info(f"Codigo: {lista_codigo}")
logger.info(f"Longitud del codigo: {len(lista_imgs_codigo)}")
```

Figura 58. Logs y mezclado de los códigos. Fuente: Elaboración propia

Logs para poder hacer un seguimiento del progreso del programa y mezclado de las listas de números y letras

La función nos devolverá las siguientes variables:

```
return lista_imgs_codigo, diccionario
```

Figura 59. Retornable de la función. Fuente: Elaboración propia

Salida de la función, tendremos dos variables;

1. Lista con las rutas a las imágenes que formaran el codigo, tanto de numeros como de letras.
2. Diccionario para poder tener un registro a posteriori de los numeros y letras que se han generado.

- Función **pascal_voc_to_yolo**: esta funcion nos permitirá pasar de coordenadas "xmin, ymin, xmax, ymax" de la bbox a coordenadas en formato YOLO "xcentro, ycentro, ancho, alto".

```
def pascal_voc_to_yolo(x1, y1, x2, y2, img_w, img_h):
    """Funcion que nos pasara de coordenadas xmin,
      ymin, xmax, ymax a las coordenadas de yolo
      x,y (centro de cada bbox)
      y nos normalizara esas medidas gracias a
      que tambien recibe la anchura y la altura
      de la imagen.
    Args:
        x1 (int): Coordenada x minima de la bbox
        (la de arriba a la izquierda)
        y1 (int): Coordenada y minima de la bbox
        (la de arriba a la izquierda)
        x2 (int): Coordenada x máxima de la bbox
        (la de abajo a la derecha)
        y2 (int): Coordenada y máxima de la bbox
        (la de abajo a la derecha)
        img_w (int): Ancho de la imagen 'fondo'
        img_h (int): Largo de la imagen 'fondo'
    Returns:
        lista: La funcion nos devuelve una lista
        con 4 elementos:
          1-Centro x de la bbox
          2- Centro y de la bbox
          3-Ancho de la bbox
          4-Alto de la bbox
    """
    return [((x2 + x1)/(2*img_w)), ((y2 + y1)/(2*img_h)),
            (x2 - x1)/img_w, (y2 - y1)/img_h]
```

Figura 60. Función para pasar de coordenadas absolutas

a coordenadas tipo YOLO. Fuente: Elaboración propia

- Función **leer_txt_dict**: esta función realizará la lectura del archivo .txt done tendremos todas las clases posibles con las que trabajara nuestro programa, tanto numeros como letras.

```
def leer_txt_dict(ruta):
    """Funcion que leera el texto donde tengamos
    las clases definidas
    Args:
        ruta (path): ruta al archivo de las clases
    Returns:
        diccionario: diccionario con todas las clases
        obtenidas del fichero llamado anterior
    """
    with open(ruta,'r') as file:
        lineas = file.readlines()
        clases = {}
        for n_linea, linea in enumerate(lineas):
            clases[linea[0]] = n_linea
    return clases
```

Figura 61. Función que nos leerá el archivo .txt de todas las clases. Fuente: Elaboración propia.

Esta función nos leerá todas las clases posibles, que luego seran asociadas con el diccionario de registro.

- Función **pegar_código**: función que realizará la composición (tiling) de las imágenes seleccionadas previamente para formar el código.

```
def pegar_codigo(lista_codigo, ruta_fondo, ruta_clases,
                 diccionario_clases, offset_x, cnt):
    """Funcion que nos va a pegar las imagenes de las
        letras/numeros en la foto del fondo que hayamos
        elegido y la guardara
    Args:
        lista_codigo (lista): Lista con as rutas a las
            imagenes que conformaran el codigo
        ruta_fondo (path): Ruta a la imagen que se va
            a usar como plantilla/fondo
        ruta_clases (path): Ruta al archivo que contiene
            todas las clases
        offset_x (int): interespaciado de las fotos,
            lo variaremos en funcion que la separacion
            de letras que queramos
        cnt (int): contador del numero de imagenes
    """
```

Figura 62. Breve descripción de la función que realizara la composición de imágenes. Fuente: Elaboración propia.

Declaración de variables necesarias para la función:

```
fondo = Image.open(ruta_fondo)
ancho_fondo, alto_fondo = fondo.size
ref_reescalado = min(ancho_fondo, alto_fondo)
digitos = len(lista_codigo)
proporcion = 1.5 + 0.8*digitos
clases = leer_txt_dict(ruta_clases)
id = []
bbox_yolo = []
id_borrado = []
cnt_borrado = 0
```

Figura 63. Declaración de variables necesarias. Fuente: Elaboración propia.

- Abrimos la imagen del fondo
- Establecemos su ancho y su alto
- Declaramos una referencia de reescalado con el mínimo entre el ancho y el alto del fondo para que todas las imágenes de los recortes sean cuadradas.
- Dígitos: longitud de la lista del código
- Proporción: expresión matemática de una recta: $y = mx + n$ que nos servirá para escribir cada código en lugares diferentes de la imagen de fondo.
- Clases: total de clases que pueden existir, llamaremos a la función leer_txt_dict para que nos lea el archivo “classes.txt”.
- Id: lista con la id (1, 4, S, v..) de cada elemento que formara el código.
- Bbox_yolo: bbox en formato YOLO (centro x, centro y, ancho, alto) con respecto a la imagen del fondo de los recortes que formaran el código.
- Id_borrado: lista que nos almacenara la id de los recortes que no entran en la imagen de fondo para posteriormente borrarlos y que no se generen etiquetas “fantasma”.
- Cnt_borrado: Inicializamos un contador para tener un registro de cuantas imágenes se han borrado.

Bucle sobre todos los elementos del código y declaración de medidas y proporciones:

```
for index, imagen in enumerate(lista_codigo):
    foto = Image.open(imagen)
    ancho_foto, alto_foto = foto.size
    ratio = ref_reescalado/ancho_foto
    ancho_foto = int(ancho_foto*ratio/proporcion)
    alto_foto = int(alto_foto*ratio/proporcion)
    foto = foto.resize((ancho_foto, alto_foto))
    bbox = foto.getbbox()
    limite_x = random.randint(0, ancho_fondo - ancho_foto)
    limite_y = random.randint(0, alto_fondo - alto_foto)
```

*Figura 64. Bucle principal de la función **pegar_código.** Fuente: Elaboración propia.*

Iteramos tantas veces como elementos vayamos a tener en el código, en cada iteración se realizarán las siguientes operaciones:

- Foto: se abrirá la foto correspondiente a la posición de la lista “lista_codigo”.
- Se establecerán el ancho y el alto de la imagen
- Ratio: ref_reescalado/ancho_foto: Se calcula la ratio de reescalado dividiendo la variable ref_reescalado entre la anchura de la imagen actual.
- Ancho_foto: Se calcula el nuevo ancho de la imagen reescalada.
- Alto_foto: Se calcula el nuevo alto de la imagen reescalada .
- Foto: Se redimensiona la imagen actual utilizando los valores de ancho y alto calculados previamente.
- Bbox: bbox del recorte sin normalizar.
- Limite_x: límite en el eje “x” establecido como referencia para el borrado de imágenes.
- Limite_y: límite en el eje “y” establecido como referencia para el borrado de imágenes.

Condiciones de posición del código:

```
if index == 0:
    xy_min = [bbox[0] + int(limite_x), bbox[1] + int(limite_y)]
    xy_max = [bbox[2] + int(limite_x), bbox[3] + int(limite_y)]
else:
    if xy_max[0] >= ancho_fondo - ancho_foto:
        lista_id_borrado = imagen.split("/")
        id_borrado.append(lista_id_borrado[-2])
        if diccionario_clases[id_borrado[cnt_borrado]] == 0:
            pass
        else:
            diccionario_clases[id_borrado[cnt_borrado]] -=1
        cnt_borrado += 1
        continue
    xy_min = [xy_min[0] + ancho_foto + (offset_x), xy_min[1]]
    xy_max = [xy_max[0] + ancho_foto + (offset_x), xy_max[1]]
```

Figura 65. Condiciones para la posición del código y el borrado de letras en caso de que alguna no entre en la imagen de fondo. Fuente: Elaboración propia.

Condición para evaluar si es la primera imagen del código, si lo es, definiremos los limites mínimos y máximos donde empezara a componerse el código.

Lista_id_borrado: será una lista con las divisiones de la ruta a la imagen del estilo: [dataset, letras, A, A.jpg].

Id_borrado: lista de registro de las ids pertenecientes a las fotos eliminadas del código.

En caso de que no sea la primera imagen del código, se evaluará si la imagen que se pretende escribir entra en la imagen de fondo teniendo en cuenta los límites establecidos y se saltará esa imagen en caso de que no los cumpla y se calcularan las coordenadas de la siguiente foto para su posterior composición.

Conversión de coordenadas de la bbox a formato YOLO:

```
lista_id = imagen.split("/")
bbox_yolo.append(pascal_voc_to_yolo(xy_min[0],xy_min[1],
                                    xy_max[0], xy_max[1],
                                    ancho_fondo, alto_fondo))
fondo.paste(foto, (xy_min[0], xy_min[1]))
id.append(lista_id[-2])
```

Figura 66. Transformado a coordenadas tipo YOLO y pegado de la imagen de la letra / número. Fuente: Elaboración propia.

Lista_id: será una lista con las divisiones de la ruta a la imagen del estilo: [dataset, letras, A, A.jpg].

A continuación, normalizaremos y pasaremos a formato YOLO las bbox de las imágenes recortadas con respecto a las dimensiones del fondo.

Finalmente, pegaremos la imagen “foto” (correspondiente con el recorte reescalado) en la imagen de fondo en las coordenadas que se observan en la imagen e iremos rellenando la lista “id” para tener registro de los elementos que formarán nuestro código.

Generación de etiquetas y logs:

```
now = datetime.now()
now.strftime("%d/%m/%Y_%H:%M:%S")
fondo.save(f"../results/images/codigo_{str(cnt)}_{now}.jpg", "JPEG")
for num in range(len(id)):
    guardar_txt(id[num],clases, bbox_yolo[num], cnt, now)

logger.info(f"Elementos que formaran el codigo: {id}")
logger.info(f"Lista de elementos para borrar: {id_borrado}")
logger.info("\nSIGUENTE IMAGEN\n")
```

Figura 67. Información que se dará por terminal para ver el progreso del programa. Fuente: Elaboración propia.

Una vez fuera del bucle anterior guardaremos las imágenes con un nombre conveniente para su identificación y con objetivo de que no se solapen si generamos más.

Iniciaremos un bucle iterando por el número de elementos del código y llamaremos a la función **guardar_txt** que nos generará las etiquetas en formato YOLO de cada recorte.

Finalmente se mostrarán por pantalla información útil para ver el progreso del programa.

- Función **guardar_txt**: esta función será la responsable de la generación de las etiquetas por cada imagen.

```
def guardar_txt(id, clases,  bbox, cnt, now):
    """Fucion que nos va a generar etiquetas en formato yolo por cada imagen
    Args:
        id (str): Cada letra / numero que contiene el codigo
        clases (diccionario): Diccionario con el archivo classes.txt
        bbox (lista de listas): Lista de listas con las coordenadas
            de las bboxes en formato yolo de cada imagen que forma el codigo
        cnt (int): contador de imagenes
    """
    with open(f"../results/labels/codigo_{str(cnt)}_{now}.txt","a") as f:
        f.write(f'{clases[id]} {bbox[0]} {bbox[1]} {bbox[2]} {bbox[3]}\n')
```

*Figura 68. Función **guardar_txt** para la generación de etiquetas. Fuente: Elaboración propia.*

La generación de etiquetas se realizará imprimiendo línea a línea en un archivo ".txt" la **id de la clase** (correspondiente con la posición de esa clase en el archivo **classes.txt**), y las coordenadas de la **bbox** que albergarán cada número.

- Función **generar_diccionario**: función que creara el diccionario para la posterior interpretación de los resultados.

```python
def generar_diccionario():
    """Funcion que nos generará el diccionario con el
    registro de los numeros y letras creados en los codigos"""
    numeros_list = [str(num) for num in range(10)]
    abecedario = (list(string.ascii_uppercase) +
                  list(string.ascii_lowercase))
    keys = numeros_list + abecedario
    diccionario_clases = {key:0 for key in keys}
    return diccionario_clases
```

Figura 69. Función que genera el diccionario para obtener un registro de las clases generadas. Fuente: Elaboración propia.

Esto se hará declarando dos listas, una con números del 0 al 9, otra con las letras del abecedario, tanto en minúsculas como en mayúsculas, se juntarán ambas listas y se creara el diccionario final inicializando todas las clases a "0" en primera instancia.

11.3 Funcionamiento

Para llevar a cabo el funcionamiento de este proyecto tendremos que realizar los siguientes pasos:

1. **Levantamiento de contenedores (Dockers).**

   Como se mencionó anteriormente, este proyecto hace uso de tecnología **Docker** para su ejecución, por lo que primeramente hay que poner en funcionamiento esos contenedores para que todo puedo ejecutarse correctamente.

   Para ello, una vez estemos situados en la carpeta del repositorio, ejecutaremos el siguiente comando en caso de que sea la primera vez que levantamos los contenedores:

```
docker-compose up --build
```

Figura 70. Comando para primer levantamiento de contenedores. Fuente: Elaboración propia.

   En caso de que no sea la primera vez que los levantamos ejecutaremos el siguiente comando:

```
docker-compose up
```

Figura 71. Comando para el levantamiento normal de contenedores. Fuente: Elaboración propia.

   El comando ***docker-compose up*** lo que hace es levantar los servicios existentes en el archivo **docker-compose.yml** (Figura 27).

   El añadido ***--build*** se utiliza para que, en caso de que sea la primera vez que se levantan los servicios, se ejecute el archivo **Dockerfile (**Figura 28. Archivo Dockerfile donde se encontrarán las instrucciones, directorios y requerimientos necesarios para el correcto funcionamiento del contenedor.

2. Interacción con la API

Una vez ejecutado el comando anterior, en la terminal aparecerá la dirección donde se están levantando los contenedores (previamente definida en el archivo **docker-compose.yml**.

```
daniroldan@daniroldanpc:~/github/utiles/OCR_dataset_generator$ docker-compose up
[+] Running 1/0
 ⠿ Container ocr_dataset_generator-api-1  Created                          0.0s
Attaching to ocr_dataset_generator-api-1
ocr_dataset_generator-api-1  | INFO:     Uvicorn running on http://0.0.0.0:5000 (Press CTRL+C to quit)
ocr_dataset_generator-api-1  | INFO:     Started reloader process [1] using statreload
ocr_dataset_generator-api-1  | INFO:     Started server process [8]
ocr_dataset_generator-api-1  | INFO:     Waiting for application startup.
ocr_dataset_generator-api-1  | INFO:     Application startup complete.
```

Figura 72. Salida por terminal del funcionamiento de la API. Fuente: Elaboración propia.

Ingresaremos en el enlace y escribiremos en la barra de direcciones "/docs" seguido de la dirección para acceder a la API.

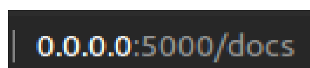


Figura 73. URL a la API. Fuente: Elaboración propia.

Esta interfaz se llama **Swagger UI** y permite a los desarrolladores interactuar con la API, ver la documentación y realizar pruebas en vivo.

Figura 74. Pantalla principal de la API. Fuente: Elaboración propia.

Como podemos ver, la API va a contar con 3 métodos

Para utilizar cualquiera de ellos, tendremos que desplegarlos y darle al botón de **try it out**.

a. **Healthcheck**: método para comprobar que los contenedores están levantados (esto también se puede comprobar por terminal).

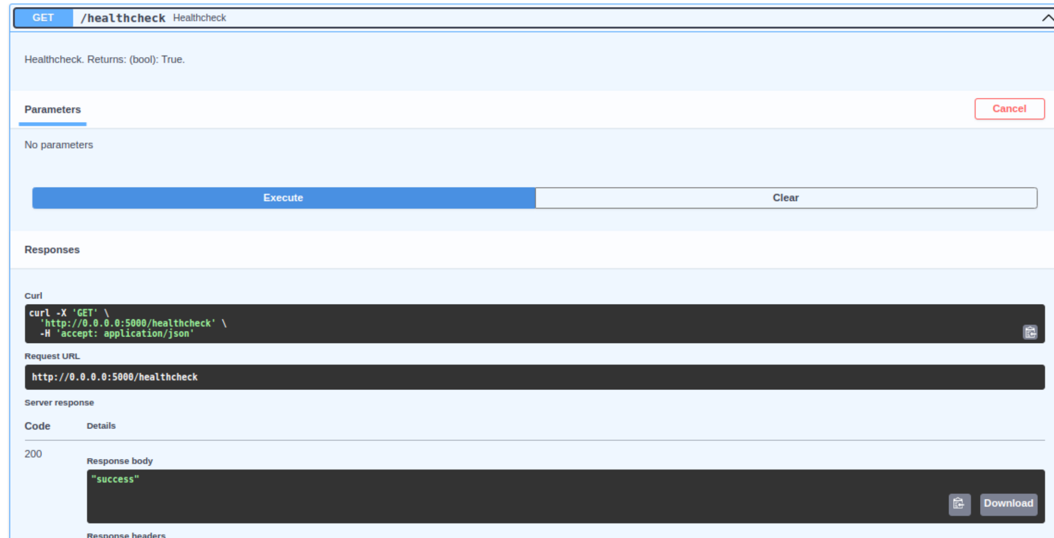


Figura 75. Pantalla principal de la API. Fuente: Elaboración propia.

Este script se utilizará dando al botón de "execute", esto nos devolverá un "success" en caso de funcionamiento correcto

b. **Dataset_image_parameters**: método utilizado para generar un dataset de recortes de letras y números que servirá como input al siguiente método.

Para este ejemplo, se han seleccionado los siguientes parámetros

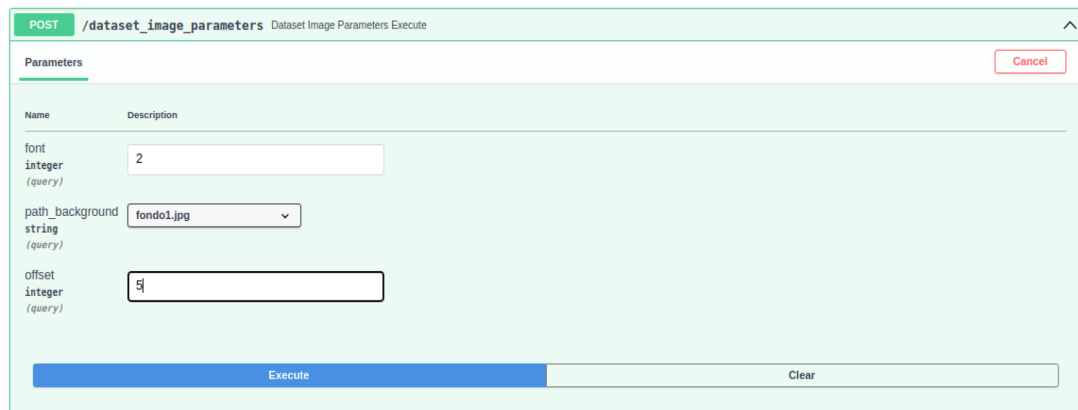


Figura 76. Método para la generación del dataset de recortes. Fuente: Elaboración propia.

Se van a generar imágenes cuya fuente corresponde con el número 2.

Hemos seleccionado el fondo número 1 de todos los fondos de los que disponemos.

Figura 77. Fondo escogido. Fuente: Elaboración propia.

En este caso se ha escogido un valor de offset = 5, es un valor intermedio.

Clicaremos en **execute** y nos fijaremos en la respuesta de la API:

Code	Details
200	Response body


```
{
  "success": true,
  "error": null,
  "data": null
}
```


*Figura 78: respuesta de la API al método **dataset_image_parameters.** Fuente: Elaboración propia.*

Como podemos ver, todo ha salido correctamente en la ejecución del método.

Este método nos crea un dataset con recortes de números y letras en la ruta la siguiente ruta:

- “OCR_dataset_generator/src_dataset/dataset”

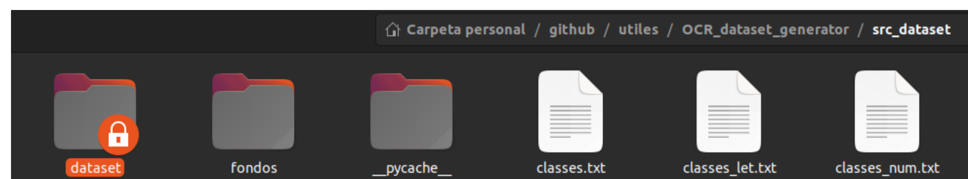


Figura 79. Carpeta generada con el dataset de los recortes. Fuente: Elaboración propia.

En esta carpeta tendremos los recortes de los números y las letras con la fuente, el fondo, y el offset escogido.

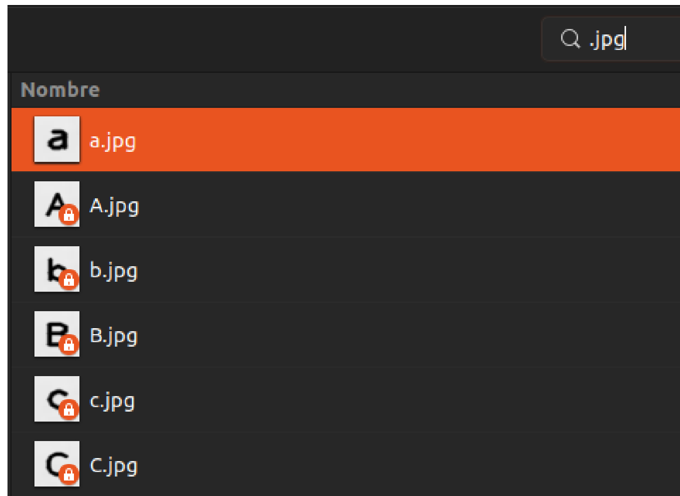


Figura 80. Ejemplo de imágenes generadas. Fuente: Elaboración propia.

c. **Previous_parameters**: método utilizado para realizar la composición de las imágenes y generación de códigos, este script tomara como datos de entrada el dataset generado anteriormente para realizar la composición aplicando la técnica de “tiling”.

POST /previous_parameters Load Parameters And Execute

Parameters

Name	Description
num_img * required integer (query)	50
num_letters * required integer (query)	4
num_numbers * required integer (query)	8
path_dataset string (query)	/src_dataset/dataset
path_classes string (query)	/src_dataset/classes.txt
path_background string (query)	fondo1.jpg
offset integer (query)	-2

Execute

Figura 81. Método para la generación del dataset de códigos. Fuente: Elaboración propia.

Como se puede observar, en este caso vamos a generar 50 imágenes con sus correspondientes 50 etiquetas.

1. Los códigos van a estar formados por un máximo de 4 letras y un máximo de 8 números, este número no es fijo ya que, al ir cambiando la posición del código dentro de la imagen de fondo, es posible que algunos elementos no entren y se tengan que eliminar.

2. La variable "path_dataset" corresponde con la ruta al dataset de entrada que se va a utilizar como fuente para la composición.

3. La siguiente variable es la ruta al archivo contenedor de todas las clases posibles para las que se ha diseñado el programa, es decir, letras de la "A" a la "Z" tanto en minúsculas como en mayúsculas y números del 0 al 9.

4. La siguiente variable corresponde con el fondo que vamos a escoger, esta variable se da en forma de desplegable y muestra todas las opciones de imágenes para fondo que tengamos en la carpeta: "OCR_dataset_generator/src_dataset/fondos".

5. Finalmente tendremos el offset, número encargado de decidir la separación entre las imágenes de los recortes que formarán el código final.

Clicaremos en el botón de "execute" y, en caso de que todo haya ido correctamente por parte de la API, nos dará la siguiente respuesta:

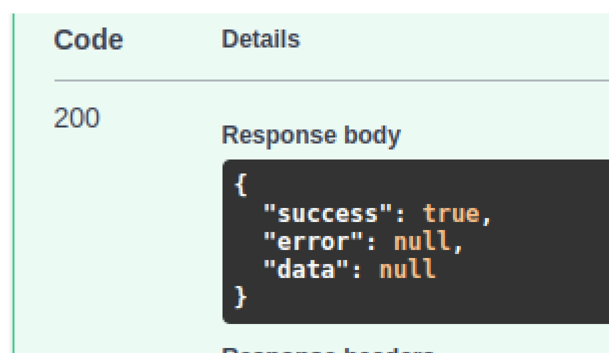


*Figura 82. Respuesta de la API al método **previous_parameters.** Fuente: Elaboración propia.*

Los resultados del programa se guardarán en la carpeta "OCR_dataset_generator/results" donde podremos ver las imágenes (50 en nuestro caso), etiquetas (50 en nuestro caso), y el archivo ".json" con el diccionario de registro de la calidad de nuestro dataset de salida.

Un ejemplo de imagen generada con su correspondiente etiqueta podría ser el siguiente:

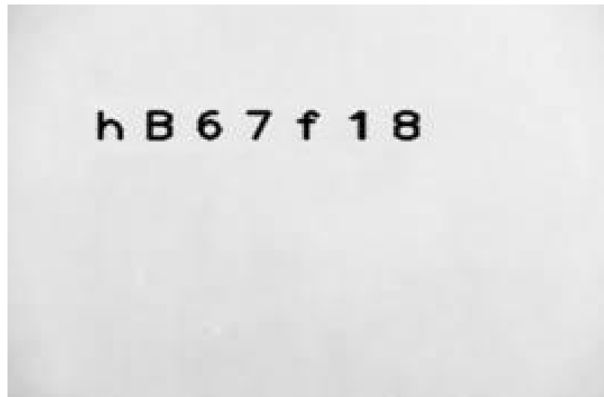


Figura 83. Ejemplo de imagen de código generado artificialmente. Fuente: Elaboración propia.

```
1 43 0.1690909090909091 0.3087431693989071 0.09090909090909091 0.1366120218579235
2 11 0.24909090909090909 0.3087431693989071 0.09090909090909091 0.1366120218579235
3 6 0.3290909090909091 0.3087431693989071 0.09090909090909091 0.1366120218579235
4 7 0.4090909090909091 0.3087431693989071 0.09090909090909091 0.1366120218579235
5 41 0.4890909090909091 0.3087431693989071 0.09090909090909091 0.1366120218579235
6 1 0.5690909090909091 0.3087431693989071 0.09090909090909091 0.1366120218579235
7 8 0.649090909090909 0.3087431693989071 0.09090909090909091 0.1366120218579235
```

Figura 84. Etiqueta correspondiente a la imagen anterior. Fuente: Elaboración propia.

La primera columna corresponde con la clase de cada elemento de la imagen.

La segunda hace referencia a la posición en el eje “x” del centro de la bbox.

La tercera hace referencia a la posición en el eje “y” del centro de la bbox.

La cuarta corresponde con el ancho de esta bbox.

La quinta corresponde con el alto de la bbox.

Cada línea del archivo de la etiqueta correspondiente a cada imagen está asociada con la bbox de cada número que forma el código, por ejemplo, en el primer lugar tenemos una “h”, si nos vamos al archivo classes.txt podremos observar que la “h” ocupa la posición 44.

```
38 b
39 c
40 d
41 e
42 f
43 g
44 h
45 i
46 j
47 k
48 l
```

*Figura 85. Archivo **classes.txt.** Fuente: Elaboración propia.*

Sin embargo, hay que tener en cuenta que el archivo classes.txt empieza a contar las líneas desde el 1, por tanto, habrá que restarle 1 a línea que ocupe la letra (44-1 = 43), número correspondiente con la etiqueta generada.

Una correcta generación de las etiquetas es muy importante para que posteriormente al entrenar el modelo, este asocie los patrones de ese número / letra de la imagen a un número / letra en formato texto.

Así podríamos ir comprobando que cada etiqueta generada se corresponde con su letra o número asociado en el archivo de las clases.

A demás de la carpeta con los resultados, la **terminal** también nos va a ofrecer información muy útil sobre la salida del programa.

Siguiendo con la información sobre la generación de la foto del ejemplo anterior:

```
                              SIGUENTE IMAGEN

[04/05/23 19:10:23] DEBUG     Lista de numeros: ['1', '8',    utils_dataset.py:55
                              '6', '7']
[04/05/23 19:10:23] DEBUG     Longitud lista numeros: 4       utils_dataset.py:56
[04/05/23 19:10:23] DEBUG     Lista de letras: ['B', 'f',     utils_dataset.py:58
                              'h']
[04/05/23 19:10:23] DEBUG     Longitud lista letras: 3        utils_dataset.py:59
[04/05/23 19:10:23] INFO      Codigo: ['6', '1', 'B', '7',    utils_dataset.py:65
                              'h', '8', 'f']
[04/05/23 19:10:23] INFO      Longitud del codigo: 7          utils_dataset.py:66
[04/05/23 19:10:23] INFO      Elementos que formaran el      utils_dataset.py:167
                              codigo: ['h', 'B', '6', '7',
                              'f', '1', '8']
[04/05/23 19:10:23] INFO      Lista de elementos para        utils_dataset.py:168
                              borrar: []
[04/05/23 19:10:23] INFO                                     utils_dataset.py:169
                              SIGUENTE IMAGEN
```

Figura 86. Información del transcurso del programa en imagen normal. Fuente: Elaboración propia.

La terminal nos ofrecerá información sobre los números que formaran el código [6, 1, B, 7, h, 8, f], así como su longitud, también nos dará la misma información sobre las letras.

Finalmente, nos proporcionará la longitud del código final, el propio código después del mezclado [h, B, 6, 7, f, 1, 8], y las letras o números que se van a borrar porque exceden los límites de la imagen (ninguno en este caso).

Un ejemplo de borrado de letras que no entran en la imagen de fondo podría ser el siguiente:

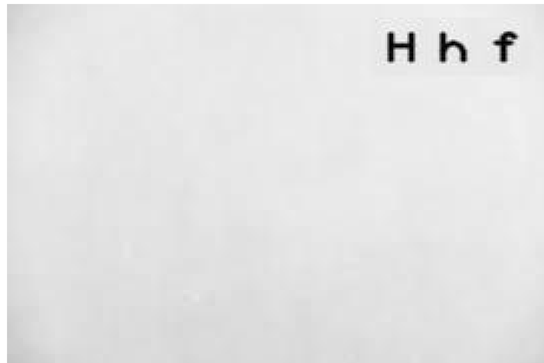


Figura 87. Ejemplo de imagen con borrado de números. Fuente: Elaboración propia.

Obtendremos la siguiente información por terminal:

```
                               SIGUENTE IMAGEN

[04/05/23 19:10:23] DEBUG    Lista de numeros: ['7', '9']    utils_dataset.py:55
[04/05/23 19:10:23] DEBUG    Longitud lista numeros: 2       utils_dataset.py:56
[04/05/23 19:10:23] DEBUG    Lista de letras: ['e', 'f',     utils_dataset.py:58
                             'h', 'H']
[04/05/23 19:10:23] DEBUG    Longitud lista letras: 4        utils_dataset.py:59
[04/05/23 19:10:23] INFO     Codigo: ['7', 'h', 'e', 'f',    utils_dataset.py:65
                             'H', '9']
[04/05/23 19:10:23] INFO     Longitud del codigo: 6          utils_dataset.py:66
[04/05/23 19:10:23] INFO     Elementos que formaran el      utils_dataset.py:167
                             codigo: ['H', 'h', 'f']
[04/05/23 19:10:23] INFO     Lista de elementos para        utils_dataset.py:168
                             borrar: ['e', '9', '7']
[04/05/23 19:10:23] INFO                                    utils_dataset.py:169
                             SIGUENTE IMAGEN
```

Figura 88. Información del transcurso del programa en imagen con borrado de elementos. Fuente: Elaboración propia.

Como vemos, el código iba a ser: [7, h, e, f, H, 9] después del mezclado se vería de la siguiente manera: [H, h, f, e, 9, 7] y como los últimos 3 elementos no entran en la imagen de fondo, se borran con el objetivo de no generar **etiquetas fantasma** que hagan que el modelo se confunda.

Con **etiquetas fantasma** nos referimos a etiquetas cuyas coordenadas de la bbox correspondiente al elemento se salgan de los límites de la imagen.

El diccionario de registro de letras y números generados en los códigos en el total de 50 imágenes será el siguiente:


```
1 {"0": 14, "1": 17, "2": 13, "3": 10, "4": 11, "5": 13, "6": 22, "7": 15, "8": 17,
  "9": 12, "A": 2, "B": 5, "C": 2, "D": 1, "E": 1, "F": 3, "G": 0, "H": 1, "I": 3,
  "J": 1, "K": 0, "L": 2, "M": 6, "N": 4, "O": 0, "P": 0, "Q": 0, "R": 0, "S": 2,
  "T": 0, "U": 0, "V": 3, "W": 1, "X": 0, "Y": 2, "Z": 1, "a": 3, "b": 2, "c": 1,
  "d": 0, "e": 0, "f": 4, "g": 3, "h": 5, "i": 0, "j": 3, "k": 1, "l": 2, "m": 7,
  "n": 1, "o": 2, "p": 1, "q": 1, "r": 1, "s": 0, "t": 1, "u": 0, "v": 5, "w": 2,
  "x": 1, "y": 4, "z": 1}
```


Figura 89. Diccionario de registro de clases. Fuente: Elaboración propia.

Como vemos, el dataset de salida no está muy equilibrado, ya que tiene muchos números y hay letras que no tenemos, como la “T”, “K” ...

En estos casos se aconsejaría repetir la generación del dataset de salida poniendo en la API una cantidad de letras y números más cercana (5 números, 5 letras).

12. ELECCIÓN DEL MODELO, PREPARACIÓN DEL DATASET Y ENTRENAMIENTO

Es muy importante elegir el modelo adecuado para entrenar un determinado tipo de datos, por ejemplo, no se puede escoger el mismo modelo si lo que se quiere entrenar son datos en formato imagen (modelo basado en la extracción de características visuales a través de redes neuronales convolucionales (CNN) y), que si lo que se pretende es entrenar datos en forma de audio.

12.1. Elección del modelo de Deep Learning

Como en nuestro caso tenemos que entrenar datos en formato imagen, nuestra decisión ha sido optar por uno de los modelos más potentes en cuanto a poder computacional que existe, este modelo es YOLO (You Only Look Once) en su versión 8 (YOLO v8).

Este modelo utiliza redes neuronales convolucionales (CNN) de aprendizaje profundo para el aprendizaje de patrones y la extracción de características

YOLO v8 se utiliza para tareas de detección de objetos, clasificación de imágenes y segmentación de instancias.

El principal motivo de la elección de este modelo ha sido que nos enfrentamos a un problema de detección y gracias dada su rapidez y baja cantidad de datos necesarios para obtener buenos resultados se convertía en nuestro mejor candidato.

12.2. Preparación del dataset

Antes de entrenar un modelo de Deep Learning, es importante preparar un buen conjunto de datos para que tengamos los mejores resultados posibles en cuanto a detecciones realizadas por el modelo.

Para ello, en nuestro caso hemos optado por añadir 4 datasets de temática OCR de dos proyectos que tenemos que supusieran un reto y nos permitiesen cubrir un rango de aplicaciones amplio en cuanto a tipo de superficie, así como dos sacados de internet:

Los elementos de los códigos de cada uno de estos datasets formarán el dataset de entrada para el método **previous_parameters**.

El dataset de la temática en función de la superficie, será generado con ayuda del método **previous_parameters**, por ejemplo, un dataset de superficie de papel, otro de superficie de metal, etc.…

La cantidad de imágenes generadas dependerá del dataset que queramos generar, al igual que la elección de la imagen de fondo.

Para este caso se han escogido una cantidad de letras y números por código igual a 5, y como dataset de entrada le pasaremos el dataset de los recortes de las letras y números.

Posteriormente se juntarán todos los dataset de distinta temática en un solo, el de entrenamiento.

En los casos en los que ya partimos de un proyecto realizado, tenemos las imágenes con los códigos completos, pero nosotros necesitamos únicamente los elementos que lo forman.

Para suplir este problema, se ha optado por el diseño de un script que nos separe los elementos de los códigos utilizando los pesos obtenidos durante el entrenamiento de dichos proyectos.

12.2.1. Script para obtención de recortes

En los casos del dataset de códigos sobre superficie de papel y de barras de metal, tuvimos que diseñar un script para obtener los recortes de los números de todas las imágenes con las que ha sido entrenado el modelo.

Esta técnica se denomina "**inferencia**": es el proceso de utilizar los pesos que obtenemos como resultado de un entrenamiento para probar la efectividad de este, en estos dos casos las detecciones van a ser perfectas ya que el modelo ha sido entrenado con las imágenes en las que vamos a hacer inferencia para obtener los recortes de los dígitos.

Esto se llevó a cabo haciendo peticiones a un método determinado de la API que tenemos en Siali para manejar Yolov8.

El script quedo de la siguiente manera:

```
1    import os
2    import requests
3    import cv2
4    from tqdm import tqdm
```

Figura 90. Importación de librerías del script para la separación de elementos. Fuente: Elaboración propia.

Importación de librerías necesarias.

1. Librería os: utilizada para el tratamiento de directorios
2. Librería requests: utilizada para hacer peticiones en ciertos métodos de la API.
3. Librería cv2 (opencv): utilizada para el tratamiento de imágenes.
4. Librería tqdm: utilizada para tener una barra de progreso visual del programa

Preparación de la respuesta:

```
params = {
    'model_name': 'cantabrialabs_v8.pt',
}
response = requests.post('http://IP:3000/models/load_model',
                         params=params)

images_list = [f for f in os.listdir("clabs") if f.endswith('.jpg')]
yolo_url = "http://IP:3000/models/predict_image_detections"

# Números válidos a detectar
numeros_validos = [str(i) for i in range(10)]
#letras validas a detectar (mayusculas)
letras_validas = [chr(i) for i in range(65, 91)]

output_dir = "dataset_clabs"
# Crear el directorio de salida si no existe
if not os.path.exists(output_dir):
    os.makedirs(output_dir)
```

Figura 91. Declaración de variables principales. Fuente: Elaboración propia.

Cargamos los pesos haciendo uso del método “load_model” de la API para posteriormente hacer inferencia (ejemplo de dataset en superficie de papel).

Listamos todas las imágenes del dataset.

Generamos la dirección del método utilizado para hacer inferencia en las imágenes del dataset, obtener detecciones, y a partir de esas detecciones recortar la imagen.

Declaramos dos listas con los números y letras que esperamos encontrar.

Generalmente en industria se suelen usar los números del 0 al 9 y las letras mayúsculas de la “A” a la “Z” excluyendo la “Ñ”.

Comprobaremos la existencia del directorio de salida, si no existe se creará.

Inicio del bucle por todas las imágenes del dataset:

```
for image_name in tqdm(images_list):
    image_path = os.path.join("clabs", image_name)
    files = {"image": open(f"{image_path}", "rb")}
    # try:
    response = requests.post(yolo_url, files=files)
    detections = response.json()["data"]
```

Figura 92. Bucle para recorrer todas las imágenes del dataset. Fuente: Elaboración propia.

Iteramos por todas las imágenes haciendo uso de la librería **tqdm** para ver el progreso del programa.

Dentro de este bucle se evaluará la respuesta del método de la API que hace inferencia sobre la imagen correspondiente a la iteración actual y obtendremos las detecciones.

Comprobaremos la existencia del directorio de salida.

Dentro del bucle anterior, iteraremos sobre las detecciones obtenidas, ya que en una misma imagen hay varias detecciones:

```
for detection in detections:
    objeto = detection['object']
    confianza = float(detection['confidence'])
```

Figura 93. Iteramos sobre detecciones de YOLO. Fuente: Elaboración propia.

Declararemos la variable "objeto" correspondiente con el número o letra que se detecte.

También declaramos la variable "confianza" correspondiente con la confianza de detección de dicho número o letra.

Condición de detección:

```
# Verificar si el objeto detectado es un número válido
if objeto in numeros_validos or objeto in letras_validas and confianza > 0.65:
    # Obtener las coordenadas de la detección
    img_original = cv2.imread(image_path)
    image_height, image_width, _ = img_original.shape
    x, y, width, height = detection['detections']
    left = int((x-width/2) * image_width)
    top = int((y-height/2) * image_height)
    right = int((x + width/2) * image_width)
    bottom = int((y + height/2) * image_height)
    # Crear una carpeta para el número si no existe
    elemento_dir = os.path.join(output_dir, objeto)
    if not os.path.exists(elemento_dir):
        os.makedirs(elemento_dir)
    img_para_recorte = img_original.copy( (variable) bottom: int
    # Guardar el recorte de la detección                    iente
    img_recortada = img_para_recorte[top:bottom, left:right]
    img_name = image_name.split('.')[0] + f'{detection["confidence"]}'+'.png'
    ruta_recorte = os.path.join(numero_dir, img_name)
    cv2.imwrite(ruta_recorte, img_recortada)
```

Figura 94. Condición que debe de cumplir la detección. Fuente: Elaboración propia.

Si el número detectado está en la lista de números validos o letras válidas y la confianza de detección es mayor a 0.65, se realizará lo siguiente:

- Se leerá la imagen y se sacaran proporciones para, posteriormente pasar de coordenadas de detección (bbox) en formato YOLO (centro x, centro y, ancho, alto) en relativo, a coordenadas de detección absolutas (xmin, ymin, xmax, ymax).
- Se comprobará la existencia del directorio para la clasificación de las detecciones y si es la primera vez que se detecta ese número o letra, se creará.
- Se recortará la imagen original haciendo uso de las coordenadas absolutas halladas anteriormente y se guardará en su carpeta correspondiente.

A continuación, se muestra un ejemplo de los datasets con los que se va a tratar:

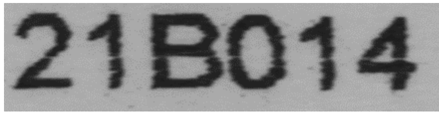


Figura 95. Dataset de entrada (proyecto de empresa)

*Figura 96. Dataset de entrada al método **previous_parameters***

Figura 97. Dataset de temática (papel en este caso)

12.3. Tipología de datasets

12.3.1. Códigos sobre superficie de papel

Este dataset consta con imágenes de códigos escritos a ordenador sobre superficie de papel, el preentrenamiento de este proyecto se ha llevado a cabo utilizando el script **generar_dataset**.

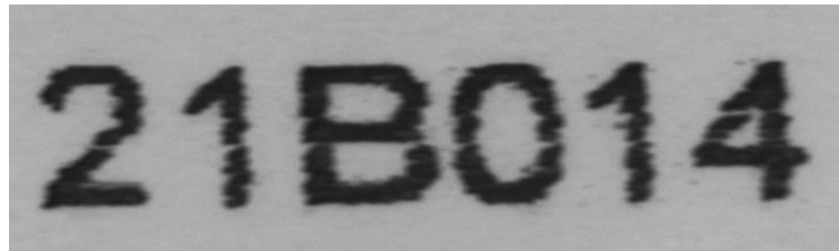


Figura 98. Imagen de los códigos de los que se extraerán los elementos. Fuente: Elaboración propia.

En este caso, necesitamos los dígitos que forman esos códigos, por ello, haremos uso del script de separación de elementos explicado anteriormente.

Para la composición de imágenes para la realización del dataset final haremos uso de la siguiente imagen de fondo:

Figura 99. Fondo para dataset de papel. Fuente: Elaboración propia.

Por último, se generó un dataset de imágenes sobre superficie de papel con un total de 400 fotos (con sus correspondientes etiquetas) haciendo uso del método **previous_parameters** de la API del generador de datasets.

Algunas imágenes de ejemplo podrían ser:

Figura 100. Imágenes resultado. Fuente: Elaboración propia.

12.3.2. Códigos sobre superficie de metal

Este dataset contiene imágenes de códigos de palanquillas (barras de metal) no escritos a ordenador.

Figura 101. Imagen de códigos sobre metal. Fuente: Elaboración propia.

Al igual que en el caso anterior, nosotros necesitamos los dígitos que forman esos códigos, por ello, haremos uso del script de separación de elementos explicado anteriormente.

Para la composición de imágenes para la realización del dataset final haremos uso de la siguiente imagen de fondo:

Figura 102. Fondo para dataset en metal. Fuente: Elaboración propia.

Por último, se generó un dataset de temática “papel” con un total de 300 imágenes (con sus correspondientes etiquetas) haciendo uso del método **previous_parameters** de la API del generador de datasets.

Algunas imágenes de ejemplo podrían ser:

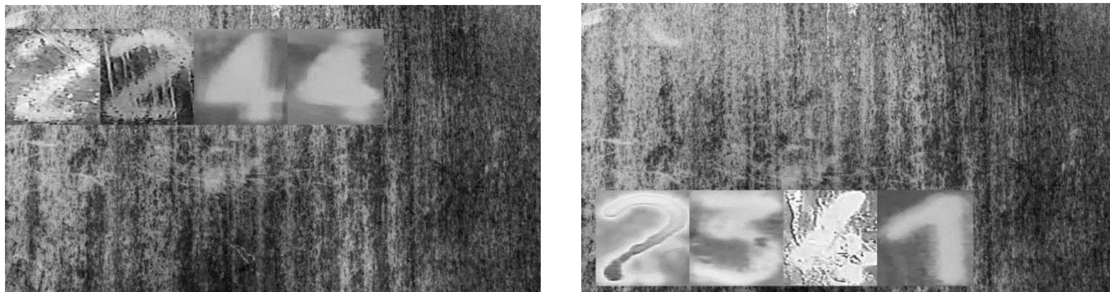


Figura 103. Ejemplos de imágenes de códigos generados en metal. Fuente: Elaboración propia.

Cabe destacar, que este dataset contaba con varios retos tener en cuenta:

- Variabilidad en cuanto a tamaño del número.
- Zonas de algún número quemadas.
- Zonas muy oscuras
- Números muy rotos en ocasiones

12.3.3. Números en 7 segmentos

Este dataset contiene números del 0 al 9 graficados en 7 segmentos sacado de internet, este dataset es muy útil si también es necesario que el modelo reconozca lo que pone en pantallas LCD.

Este dataset está compuesto por 102.000 fotos de números de pantallas LCD con variaciones, aquí se muestran algunos ejemplos.

Figura 104. Ejemplos de variaciones para dataset de 7 segmentos. Fuente: Elaboración propia.

Finalmente se generó un dataset de códigos formados por los recortes anteriores con un total de 300 imágenes (con sus correspondientes etiquetas) haciendo uso del método **previous_parameters** de la API del generador de datasets.

Se utilizó el siguiente fondo para la composición de imágenes:

Figura 105. Fondo para dataset de números 7 segmentos. Fuente: Elaboración propia.

Algunas imágenes de ejemplo podrían ser:

Figura 106. Ejemplos de imágenes generadas de números 7 segmentos. Fuente: Elaboración propia.

12.3.4. Códigos sobre metal liso

Este dataset contiene imágenes de números y letras sobre superficies metálicas lisas con cierto grado de variación.

Algunos ejemplos de imágenes podrían ser:

Figura 107. Ejemplos de imágenes con variaciones para dataset en metal liso. Fuente: Elaboración propia.

Finalmente se generó un dataset de códigos formados por los recortes anteriores con un total de 500 imágenes (con sus correspondientes etiquetas) haciendo uso del método **previous_parameters** de la API del generador de datasets.

Se utilizó el siguiente fondo para la composición de imágenes:

Figura 108. Fondo para dataset de metal liso. Fuente: Elaboración propia.

Hemos generado más imágenes que en el caso anterior con objetivo de tener número y letras lo más balanceado posible, ya que en este dataset disponemos de recortes de letras.

Algunas imágenes de ejemplo podrían ser:

Figura 109. Ejemplo de imágenes generadas para el dataset de metal liso. Fuente: Elaboración propia.

12.4. Dataset de entrenamiento

El dataset de entrenamiento va a ser una suma de todas las imágenes y las etiquetas que hemos generado anteriormente con la ayuda del método **previous_parameters** de la API.

Algún ejemplo de las imágenes de las que disponemos podría ser:

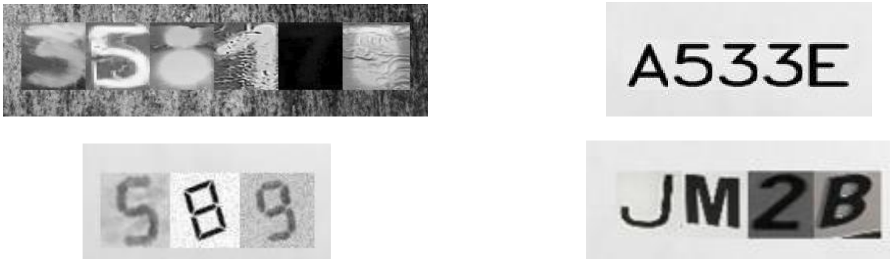


Figura 110. Ejemplo de imágenes de los datasets de las distintas superficies. Fuente: Elaboración propia.

Las imágenes deben de ir en la carpeta “images” y las etiquetas en la carpeta “labels”.

Para que un entrenamiento sea lo más completo posible y podamos obtener un “feedback” de cómo ha ido, es imprescindible separar las imágenes en dos capetas, las destinadas al entrenamiento, y las destinadas a la validación.

Generalmente se utiliza un 80% de las imágenes para entrenamiento, y un 20% para validación.

Figura 111. Distribución de carpetas para el entrenamiento. Fuente: Elaboración propia.

Dentro de estas dos carpetas tendrá que haber la siguiente estructura de subcarpetas:

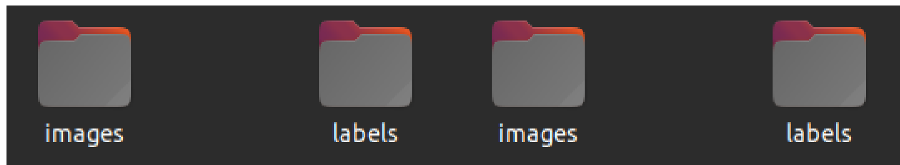


Figura 112: Distribución de imágenes y etiquetas para el entrenamiento. Fuente: Elaboración propia.

Imágenes y etiquetas de entrenamiento   Imágenes y etiquetas de validación.

A demás de las imágenes y etiquetas separadas carpetas destinadas al entrenamiento y a la validación, tenemos que añadir otro archivo esencial para que YOLO pueda realizar el entrenamiento.

Este archivo se denomina **train.yaml,** debe estar ubicado a la altura de las carpetas de entrenamiento y validación. Contiene la información necesaria para el entrenamiento como las clases y el número de estas, y las carpetas a las imágenes de entrenamiento y validación.

```
names:
- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- A
- B
- C
- D
- E
- F
- G
- H
- I
- J
- K
- L
- M
- N
- O
- P
- Q
- R
- S
- T
- U
- V
- W
- X
- Y
- Z
nc: 36
train: ./train/images/
val: ./validation/images/
```

*Figura 113. Archivo **train.yaml.** Fuente: Elaboración propia.*

Estas rutas a corresponderán con las rutas correspondientes a las imágenes dentro del contenedor donde se ejecuta YOLOv8.

12.5. Entrenamiento

Para entrenar el modelo YOLOv8, haremos uso del dataset de entrenamiento generado anteriormente contenedor de todas las imágenes y etiquetas.

En nuestro caso vamos a hacer uso de la API que tenemos en Siali para entrenar modelos de Deep Learning y hacer más sencillo el proceso.

A continuación, se explicarán los parámetros usados para este entrenamiento.

Parameters

Name	Description
base_model string (query)	yolov8m.pt
img_size integer (query)	640
epochs integer (query)	30
batch_size integer (query)	4
dataset_path string (query)	OCR_GENERAL/train.yaml

Figura 114. Parámetros con los que se realizará el entrenamiento. Fuente: Elaboración propia.

1. **Base_model:** hay distintos modelos a escoger a la hora de entrenar un modelo de YOLOv8 dependiendo de su complejidad y tamaño, en nuestro caso vamos a coger un modelo intermedio: **yolov8m.pt.**
   La elección de este modelo se debe a que este modelo ofrece una gran precision, es eficiente en cuanto a recursos computacionales y nos permite entrenar con datos complejos.

2. **Img_size:** tamaño de la imagen con el que YOLO va a trabajar, cuanta más calidad de imagen, mejores resultados, pero mayor carga de cómputo. En nuestro caso hemos elegido una calidad de imagen de 640.

3. **Epoch:** la cantidad de veces que todo el conjunto de datos de entrenamiento se ha pasado a través del modelo durante el proceso de entrenamiento. En cada época, el modelo utiliza los datos de entrenamiento para ajustar sus pesos con el objetivo de obtener la mayor precisión posible.

   Es importante tener conocimientos de Deep Learning para escoger el número adecuado para que el modelo no tenga **overfitting**.

   En nuestro y, tras varias pruebas con distinto número de épocas, hemos decidido escoger 30 épocas, ya que era el número con el que mejores resultados obteníamos.

4. **Batch_size:** este es un número que se referirá a la cantidad de ejemplos de entrenamiento que se escogerán en cada paso del modelo durante el entrenamiento, pero, es importante tener tres cosas fundamentales en cuenta a la hora de escoger este número.

    I. **Eficiencia del hardware**: un mayor número hará que el modelo tome más muestras por paso, lo que reducirá los tiempos de entrenamiento.

    II. **Uso de memoria:** El tamaño del lote afecta la cantidad de memoria necesaria para almacenar los datos de entrada. Tamaños de lote más grandes requieren más memoria, por lo que es importante considerar las limitaciones de memoria de tu sistema.

    III. **Estabilidad de entrenamiento**: El tamaño del lote también puede tener un impacto en la estabilidad del entrenamiento ya que puede coincidir que, si establecemos un valor pequeño, el modelo tome datos de muestra con mucha desviación con respecto a los demás datos dificultando la convergencia del modelo.

Como resultado del entrenamiento YOLOv8 nos proporciona las siguientes gráficas:

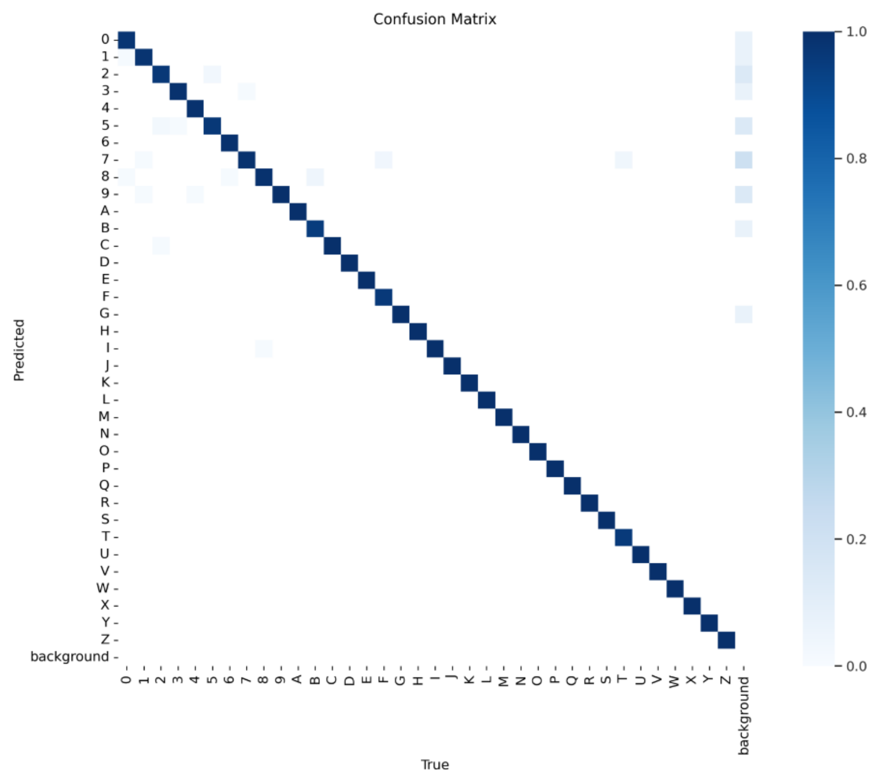


Figura 115. Matriz de confusión. Fuente: Elaboración propia.

Esta matriz la genera el modelo haciendo predicciones sobre una imagen de validación y viendo el número real que corresponde gracias a las etiquetas.

Lo ideal es que esta matriz sea lo más diagonal posible, ya que, esto indicará que el modelo apenas se ha confundido, como es nuestro caso.

Es muy útil ya que nos sirve para ver la confusión del modelo en las distintas clases, por ejemplo, arriba a la derecha podemos observar que el modelo ha hecho alguna predicción errónea, diciendo que son números, pero en realidad era fondo.

Una vez que el modelo ha sido entrenado, se generaran unos pesos (resultado del entrenamiento). Dichos pesos se utilizarán para comprobar la eficacia del modelo pasándole a este las fotos que no ha visto, este proceso se conoce como **inferencia**.

La eficacia de los pesos puede variar dependiendo de la calidad del conjunto de datos que se utilicen para el entrenamiento.

En nuestro caso, se tuvieron que realizar varios entrenamientos para dar con la solución óptima.

- En primera instancia se probó con **100 épocas** y un **batch_size** igual a **4**, pero, al evaluar el modelo observamos que no era capaz de generalizar con nuevos datos, por lo que dedujimos que había desarrollado **overfitting.**

- A continuación, probamos con los mismos ajustes, pero con menos épocas (60) y volvimos a obtener unos resultados muy similares.

- Viendo que con muchas épocas el modelo desarrollaba overfitting, probamos con 20 épocas, pero no obtuvimos unos buenos resultados de detección pese a no tener overfitting.

- Finalmente probamos con 30 épocas, y en este punto fue donde obtuvimos los mejores resultados.

13. INFERENCIA Y COMPARACIÓN CON OTROS MODELOS

Finalmente, una vez el modelo ya ha sido entrenado, aparte de observar las gráficas que nos proporciona YOLOv8, es importante que probemos el modelo con algunas imágenes que no haya visto.

En nuestro caso se va a escoger imágenes nuevas de **superficie metálica**, **papel**, alguna de **matrículas** de vehículos (como ejemplo de superficie de metal liso), y pantalla **LCD.**

Vamos a enfrentar nuestro modelo con uno de los modelos más utilizados para OCR que existe; **EasyOCR.**

Para ello, le pasaremos a cada modelo la misma imagen para que realicen las detecciones en igualdad de condiciones y podamos ver que modelo hace mejor el reconocimiento óptico de caracteres.

Es importante recalcar que las imágenes de inferencia no pueden ser imagen con las que el modelo haya entrenado, ya que, si fuese así, nos daría una precisión cercana al 100% con confianzas de detección mayores a 0.99.

Este proceso se va a realizar haciendo inferencia en ambos modelos, en nuestro caso con YOLO v8 y los pesos que hemos obtenido como resultado de entrenamiento, y en el caso de EasyOCR mediante un script.

Este script quedaría de la siguiente manera:

Importación de librerías:

```
import easyocr
import cv2
import numpy as np
import os
from tqdm import tqdm
```

Figura 116 Importación de librerías de script para probar EasyOCR. Fuente: Elaboración propia.

- Librería **EasyOCR** para utilizarla para el reconocimiento de texto en la imagen.
- Librería **opencv** para el tratamiento de imágenes.
- Librería **numpy** para generar arrays.
- Librería **os** para la definición de carpetas y directorios.
- Librería **tqdm** para ver el progreso del programa por terminal.

Programa principal:

```
# Ruta de la carpeta que contiene las imágenes a procesar
carpeta_imagenes = 'clabs'

# Llamar a la función para procesar las imágenes de la carpeta
procesar_imagenes_carpeta(carpeta_imagenes)
```

Figura 117. Programa principal. Fuente: Elaboración propia.

– El programa principal está compuesto únicamente por la declaración de la ruta a la carpeta de imágenes y la función que las procesa.

A continuación, la función que procesara las imágenes:

```
def procesar_imagenes_carpeta(carpeta):
    reader = easyocr.Reader(['en'])
    archivos = os.listdir(carpeta)

    for archivo in tqdm(archivos):
        ruta_imagen = os.path.join(carpeta, archivo)
        imagen = cv2.imread(ruta_imagen)
        imagen_gris = cv2.cvtColor(imagen, cv2.COLOR_BGR2GRAY)
        detecciones = reader.readtext(imagen_gris)
        dibujar_bboxes(imagen, detecciones)
```

Figura 118. Función para procesar imágenes. Fuente: Elaboración propia.

– Llamaremos al método **Reader** de la librería EasyOCR para que nos lea el contenido de la imagen. Le pasaremos las letras "en" para que la lectura la haga en inglés, ya que al ser códigos no que lo pongamos en inglés o español.
– Listaremos todas las imágenes que existan en la carpeta anterior.
– Iteraremos sobre todas las imágenes.
– Abrimos la imagen y la convertiremos a escala de grises porque es el formato que admite la función **Readtext** del método **Reader.**
– Llamaremos a la función **dibujar_bboxes** y le pasaremos la imagen y las detecciones. Las detecciones tienen la siguiente estructura:

[([[0, 0], [207, 0], [207, 61], [0, 61]], '221589', 0.6343861090028153)]

Siendo las 4 primeras posiciones las correspondientes a las coordenadas de la bbox que alberga todo el código, a continuación, el número detectado, y finalmente la confianza de detección del conjunto de elementos.

Finalmente, definiremos la función que nos dibujará las bboxes:

```
def dibujar_bboxes(imagen, detecciones):
    for bbox, texto, confianza in detecciones:
        bbox = [[int(x), int(y)] for x, y in bbox]
        cv2.polylines(imagen, [np.array(bbox)], True, (0, 255, 0), 2)
        cv2.putText(imagen, f'{texto}: {confianza:.2f}', (bbox[0][0], bbox[0][1] - 10),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0, 0, 255), 2)

    cv2.imshow('Imagen con Bounding Boxes', imagen)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

Figura 119. Función que nos permitirá observar las detecciones del modelo. Fuente: Elaboración propia.

– Iteraremos sobre las detecciones que ha realizado EasyOCR y se almacenara en la variable **bbox** para su posterior dibujado.

- Dibujamos las líneas que conformaran la bbox en la imagen con la ayuda de la función **polylines** de la librería opencv.
- Nos apoyaremos también en la función **putText** de la librería opencv para pintar la confianza en la imagen 10 pixeles más arriba que la bbox.
- Finalmente mostraremos la imagen resultante por pantalla.

13.1. Inferencia en imágenes con superficie metálica

Foto que les vamos a pasar a ambos modelos:

Figura 120. Imagen sobre superficie de metal para probar ambos modelos. Fuente: Elaboración propia.

En el caso del dataset de códigos obre superficie de metal, fue todo un reto afrontar estas imágenes ya que los números podían llegar de todas las posibilidades imaginables, haciendo realmente difícil su detección y etiquetado.

Si se hubiese contado con la herramienta descrita en este TFG, podríamos haber llegado a simplificar el proceso en gran medida.

- **EasyOCR**

  Ejecutamos el script y obtenemos los siguientes resultados:

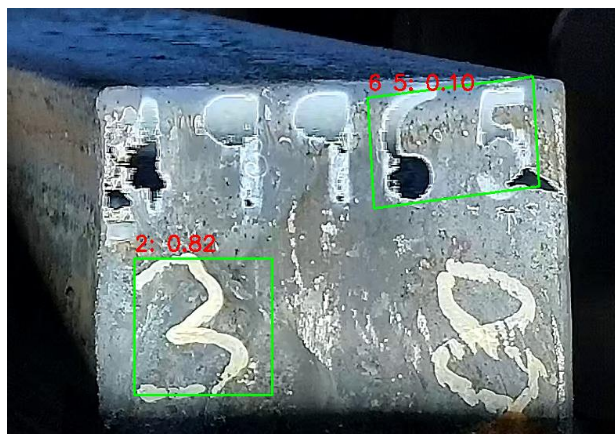


Figura 121. Resultados de detección de EasyOCR sobre superficie metálica. Fuente: Elaboración propia.

Como podemos observar las detecciones realizadas por EasyOCR en este caso no son nada precisas ya que únicamente detecta los números 6 y 5 correctamente, pero con una confianza muy baja: 0.1

También se puede apreciar cómo el modelo detecta un 2 con una confianza de 0.82 (confianza alta) a pesar de ser un 3.

- **Nuestro modelo**

Con la ayuda del método **predict_image** obtenemos los resultados de la inferencia de YOLO en la imagen anterior.

Figura 122. Resultados de detección de nuestro modelo sobre superficie metálica. Fuente: Elaboración propia.

Como no es suficiente para apreciar las detecciones, haremos uso del método que nos da los resultados de la inferencia en formato ".json" (**predict_image_detections**).

```
"data": [
  {
    "object": "4",
    "confidence": "0.8925064802",
    "detections": [
      0.24508305958340776,
      0.2642574912623684,
      0.036854471479129465,
      0.11593077810184212]},
```

```
{
  "object": "9",
  "confidence": "0.9098823071",
  "detections": [
    0.28908086958385415,
    0.2608975059107895,
    0.03926922026134673,
    0.11731593483375]},
```

```
{
  "object": "9",
  "confidence": "0.8921924233",
  "detections": [
    0.3301172710600446,
    0.2540181611713816,
    0.0384213810875372,
    0.11046172694157895]},
```

```
{
  "object": "6",
  "confidence": "0.8995550275",
  "detections": [
    0.3731741678146949,
    0.2521231801886184,
    0.0350861322312128,
    0.10900991339434211]},
```

```
{
  "object": "5",
  "confidence": "0.8660322428",
  "detections": [
    0.4178335553123884,
    0.2510567112973026,
    0.033800215948214286,
    0.11149567051940788]},
```

```
{
  "object": "3",
  "confidence": "0.8959544897",
  "detections": [
    0.2666276295979911,
    0.4017800582082895,
    0.06637091863723958,
    0.13796175906532895]},
```

```
  {
    "object": "8",
    "confidence": "0.8887782097",
    "detections": [
      0.40963686080204614,
      0.41310842413651316,
      0.06550916035970983,
      0.14944453992342105]},
]
```


Figura 123. Imágenes de las detecciones de nuestro modelo en formato ".json" sobre superficie metálica. Fuente: Elaboración propia.

Como podemos observar, nuestro modelo detecta correctamente todos los números y con una confianza superior a 0.86 por lo que podemos afirmar que el modelo está muy seguro de lo que está detectando.

13.2. Inferencia en imágenes con superficie de papel

Foto que le vamos a pasar a ambos modelos:

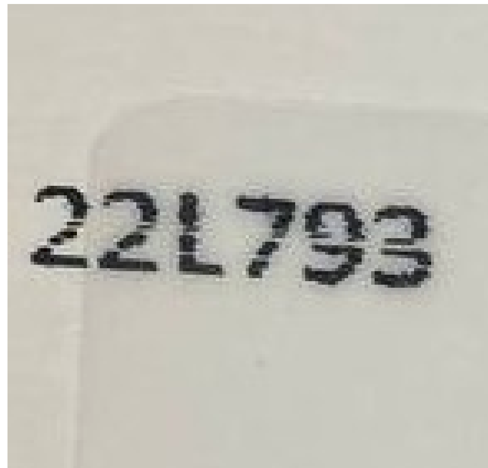


Figura 124. Imagen sobre superficie de papel para probar ambos modelos. Fuente: Elaboración propia.

- **Easyocr**

  Ejecutamos el script y obtenemos los siguientes resultados:

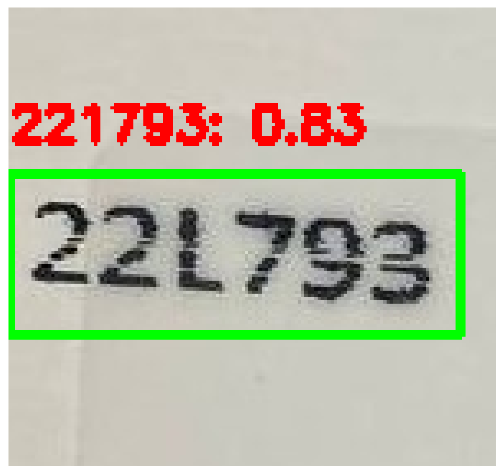


Figura 125. Resultados de detección de EasyOCR sobre superficie de papel. Fuente: Elaboración propia.

Como se puede observar, obtenemos unos resultados de detección muy prometedores (221793) todo ello con una confianza de un 0.83, a excepción de la letra “L” que la detecta como un “1”. Por tanto, es probable que hubiese que corregir el modelo e introducir más tipos de letras “L” para que no se produjese esta confusión.

- **Nuestro modelo**

  Para este caso nos hemos apoyado en la API de Yolov8 de Siali, concretamente en el método **predict_image** obteniendo los siguientes resultados con los pesos que se han obtenido en el entrenamiento:

Figura 126. Resultados de detección de nuestro modelo sobre superficie de papel. Fuente: Elaboración propia.

Como no se pueden apreciar muy bien las detecciones y la confianza, vamos a hacer uso de otro método llamado **predict_image_detections** que nos proporcionará las detecciones en formato texto.


```
"data": [
  {
    "object": "2",
    "confidence": "0.94089925",
    "detections": [
      0.0887899398803711,
      0.4760394287109375,
      0.17644365575929352,
      0.19420806884765626]},
```

```
  {
    "object": "L",
    "confidence": "0.9258712",
    "detections": [
      0.387913583919702,
      0.4858917236328125,
      0.13231413885457627,
      0.1859100341796875]},
```

```
  {
    "object": "2",
    "confidence": "0.9094881",
    "detections": [
      0.2466453905926635,
      0.4821163431803385,
      0.17706193987107435,
      0.19198321024576823]},
```

```
  {
    "object": "7",
    "confidence": "0.8822453",
    "detections": [
      0.5161670406922599,
      0.4960255432128906,
      0.15734605599712853,
      0.18171775817871094]},
```

```
  {
    "object": "9",
    "confidence": "0.6833163",
    "detections": [
      0.6538017222423427,
      0.5036399841308594,
      0.16672556289773904,
      0.18528485616048176]},
```

```
  {
    "object": "3",
    "confidence": "0.4085837",
    "detections": [
      0.8055894864316018,
      0.5169303894042969,
      0.2049452926938897,
      0.20058278401692708]}]
```


Figura 127. Imágenes de las detecciones de nuestro modelo en formato ".json" sobre superficie de papel. Fuente: Elaboración propia.

Como podemos observar todas las detecciones están correctamente realizadas, formando el código 22L793 (no se encuentran ordenadas) siendo la menor confianza la del número "3" con un 0.4.

13.3. Inferencia en imágenes con superficie de metal lisa

Como ejemplo de esas imágenes tomamos imágenes de matrículas de algunos coches.

Foto que le vamos a pasar a ambos modelos:

Figura 128. Imagen sobre superficie de metal liso para probar ambos modelos. Fuente: Elaboración propia.

- **Easyocr**

  Ejecutamos el script anteriormente explicado:

Figura 129. Resultados de detección de EasyOCR sobre superficie de metal liso. Fuente: Elaboración propia.

Podemos ver como si hace la detección del código completo de manera correcta (0924 BFC), en cambio, únicamente lo detecta con una confianza de 0.57, y esto, en Deep Learning suele ser una confianza de detección baja.

- **Nuestro modelo**

Haciendo uso del método de la API de YOLOv8 mencionado en el caso anterior obtenemos las siguientes detecciones:

Figura 130. Resultados de detección de nuestro modelo sobre superficie metálica lisa. Fuente: Elaboración propia.

Como no se puede diferenciar bien ni las detecciones, ni la confianza de detección, evaluaremos la respuesta de la API.


```
"data": [
    {
      "object": "0",
      "confidence": "0.93813217",
      "detections": [
        0.42813911437988283,
        0.6167668660481771,
        0.04437580108642578,
        0.06685256958007812]},
```

```
{
  "object": "9",
  "confidence": "0.8736807",
  "detections": [
    0.4638786792755127,
    0.6178874969482422,
    0.04135885238647461,
    0.06509653727213542]},
```

```
{
  "object": "2",
  "confidence": "0.96562415",
  "detections": [
    0.49793004989624023,
    0.6205446879069011,
    0.04209136962890625,
    0.06416066487630208]},
```

```
{
  "object": "4",
  "confidence": "0.92873824",
  "detections": [
    0.5346023559570312,
    0.621719233194987,
    0.04502458572387695,
    0.06404800415039062]},
```

```
{
  "object": "B",
  "confidence": "0.8974545",
  "detections": [
    0.5736476898193359,
    0.622586186726888,
    0.045815467834472656,
    0.06332041422526041]},
```

```
{
  "object": "F",
  "confidence": "0.80258214",
  "detections": [
    0.6095926284790039,
    0.6228527704874675,
    0.041045761108398436,
    0.06310997009277344]},
```

```
  {
    "object": "C",
    "confidence": "0.9149274",
    "detections": [
      0.6467148780822753,
      0.6223145167032877,
      0.04608612060546875,
      0.06489791870117187]}
]
```


Figura 131. Imágenes de las detecciones de nuestro modelo en formato ".json" sobre superficie metálica lisa. Fuente: Elaboración propia.

Como se puede observar, esta vez YOLO ha hecho las detecciones en orden, detectando cada uno de los elementos del código correctamente sin bajar la confianza de detección de 0.8, lo que nos indica que el modelo está muy seguro detectando en este caso.

13.4. Inferencia sobre pantallas LCD (numeración 7 segmentos)

Imagen que le vamos a pasar a ambos modelos:

Figura 132. Imagen sobre pantalla LCD (números 7 segmentos) para probar ambos modelos. Fuente: Elaboración propia.

- **Easyocr**

  Ejecutando el script mencionado en los anteriores apartados obtenemos los siguientes resultados:

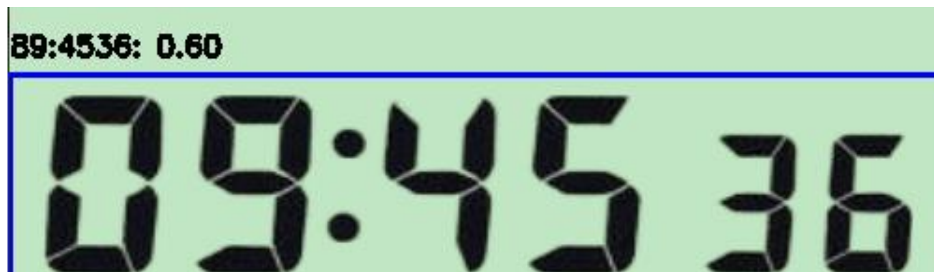


Figura 133. Resultado de detección de EasyOCR sobre pantalla LCD. Fuente: Elaboración propia.

  Como podemos ver, las detecciones realizadas por Easyocr no son del todo malas, a excepción de que detecta un número "8" cuando en realidad hay un "0" y la confianza de detección no es muy alta.

- **Nuestro modelo**

  Al igual que en los apartados anteriores, haremos uso del método **predict_image** de la API de YOLOv8.

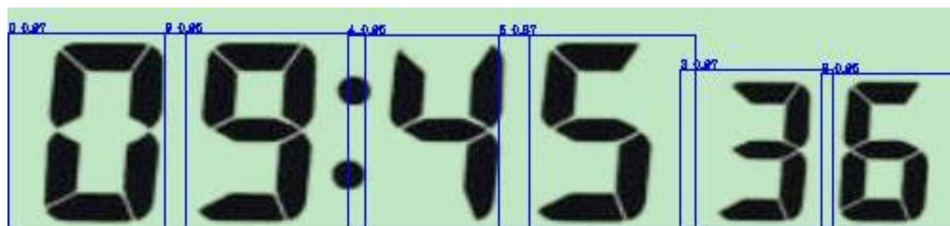


Figura 134. Resultado de detección de nuestro modelo sobre pantalla LCD. Fuente: Elaboración propia.

De nuevo, como no se aprecia muy buen las detecciones y su confianza, haremos uso del método **predict_image_detections** que nos devuelve las detecciones en formato ".json".


```
"data": [
  {
    "object": "3",
    "confidence": "0.97126615",
    "detections": [
      0.784950927734375,
      0.719083506266276,
      0.16107110595703125,
      0.5616073608398438]},
```

```
{
  "object": "0",
  "confidence": "0.96838456",
  "detections": [
    0.09400655364990235,
    0.6562729899088542,
    0.18662934875488282,
    0.6874540710449218]},
```

```
{
  "object": "9",
  "confidence": "0.9531218",
  "detections": [
    0.2695433349609375,
    0.6566213480631511,
    0.21025689697265626,
    0.6867573038736979]},
```

```
{
  "object": "4",
  "confidence": "0.9514712",
  "detections": [
    0.45228729248046873,
    0.6566845194498698,
    0.18958016967773436,
    0.6848945109049479]},
```

```
{
  "object": "6",
  "confidence": "0.94501406",
  "detections": [
    0.92611181640625,
    0.7228539021809895,
    0.14777642822265624,
    0.5501936848958333]},
```

```
    {
      "object": "5",
      "confidence": "0.8721749",
      "detections": [
        0.6179115600585937,
        0.6583202616373698,
        0.204234130859375,
        0.6826117960611979]}
  ]
}
```


Figura 135. Imágenes de las detecciones de nuestro modelo en formato ".json" sobre pantalla LCD. Fuente: Elaboración propia.

Como se puede apreciar en las imágenes de las detecciones, nuestro modelo detecta todos los números con una confianza mayor a 0.87, lo que quiere decir que el modelo está muy seguro de lo que está detectando en este caso.

13.5. Inferencia en múltiples imágenes

Con objetivo de obtener una métrica que nos indique un porcentaje de mejora de nuestro modelo con respecto al de EasyOCR se ha llevado a cabo la inferencia en **50 imágenes de cada dataset** para probar la eficacia de ambos modelos

La evaluación de los modelos se realizará de la siguiente manera:

- **Aciertos**: contará como acierto si el modelo no se ha equivocado en ningún elemento del código.
- **Fallos:** contará como fallo en el momento en el que el modelo se equivoque en la detección de algunos de los elementos que forman el código, o detecte elementos ajenos al código a tratar.

13.5.1. Códigos sobre superficie de metal

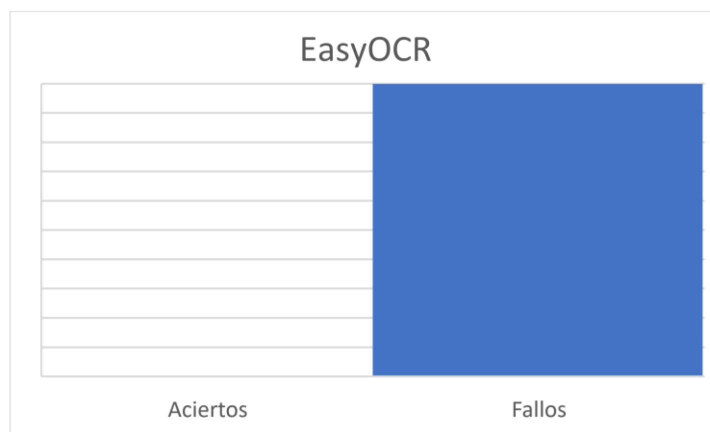


Figura 136. Aciertos en detecciones de EasyOCR para superficie de metal (0/50 aciertos). Fuente: Elaboración propia.

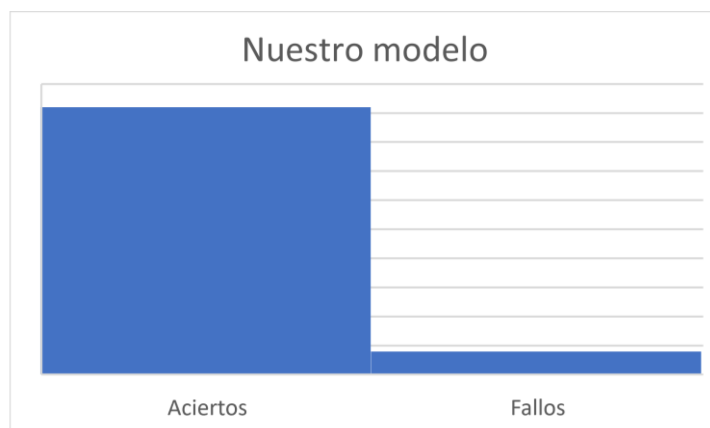


Figura 137. Aciertos en detecciones de nuestro modelo para superficie de metal (46/50 aciertos). Fuente: Elaboración propia.

Como podemos observar, nuestro modelo supera con creces a EasyOCR en este caso, ya que únicamente de ha obtenido una detección errónea en 4 ocasiones, acertando las 46 restantes.

13.5.2. Códigos sobre superficie de papel:

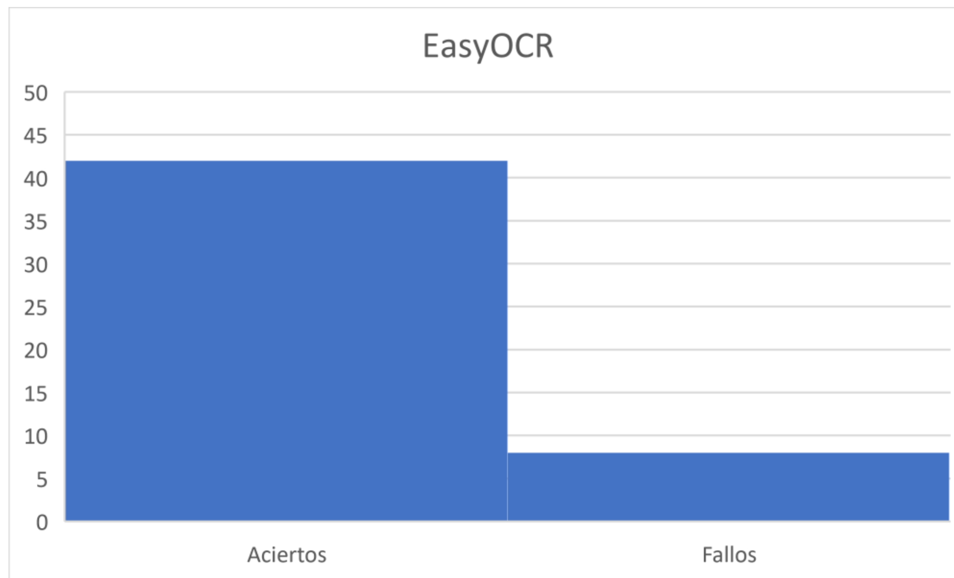


Figura 138. Aciertos en detecciones de EasyOCR para superficie de papel (42/50 aciertos). Fuente: Elaboración propia.

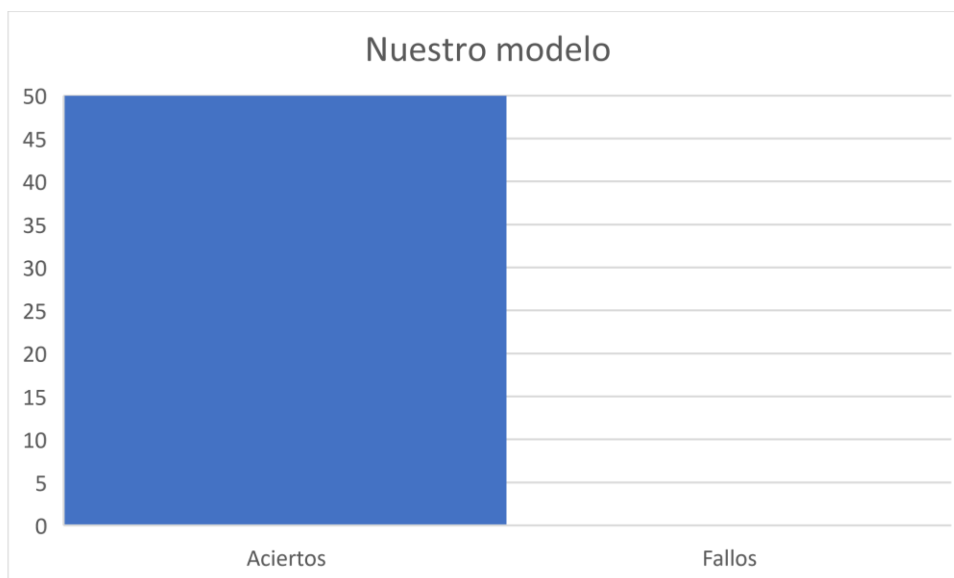


Figura 139. Aciertos en detecciones de nuestro modelo para superficie de papel (50/50 aciertos). Fuente: Elaboración propia.

En este caso se puede observar que, a pesar de que EasyOCR no lo ha hecho mal, nuestro modelo ha acertado las detecciones de las 50 imágenes con las que se ha evaluado, mientras que EasyOCR ha acertado en 42 de ellas.

13.5.3. Códigos sobre pantalla LCD:

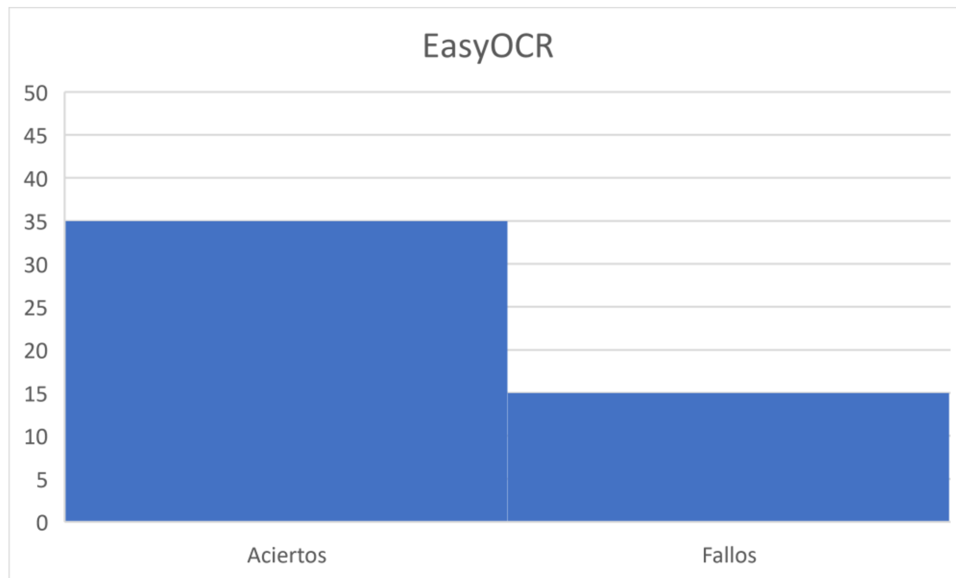


Figura 140. Aciertos en detecciones de EasyOCR para pantalla LCD (35/50 aciertos). Fuente: Elaboración propia.

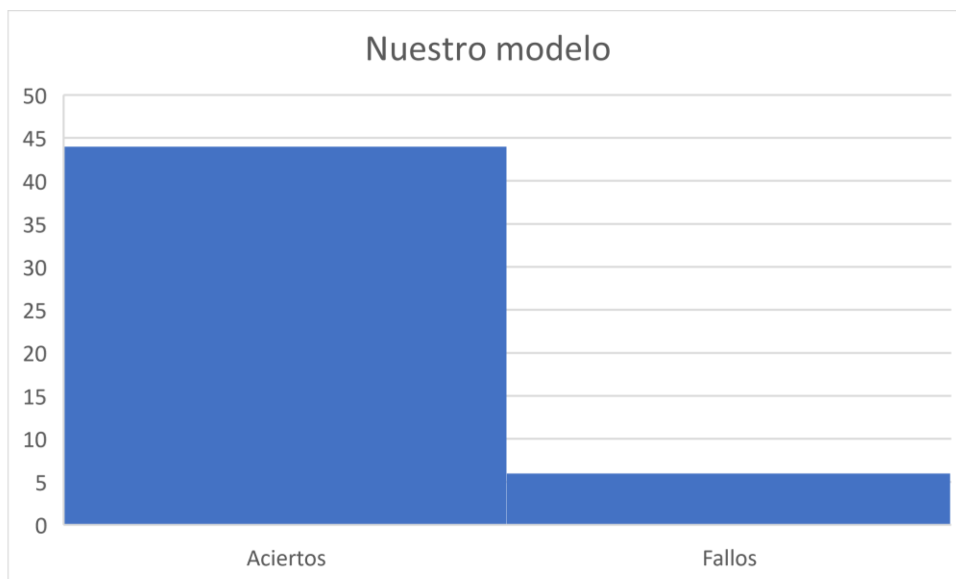


Figura 141. Aciertos en detecciones de nuestro modelo para pantalla LCD (42/50 aciertos). Fuente: Elaboración propia.

En este caso se puede apreciar como ambos modelos han tenido algún fallo y, aunque EasyOCR ha acertado menos detecciones, nuestro modelo tampoco ha sido perfecto, lo que nos dice que quizá habría que enriquecer más la parte del dataset de imágenes de pantalla LCD.

13.5.4. Códigos sobre superficie de metal liso:

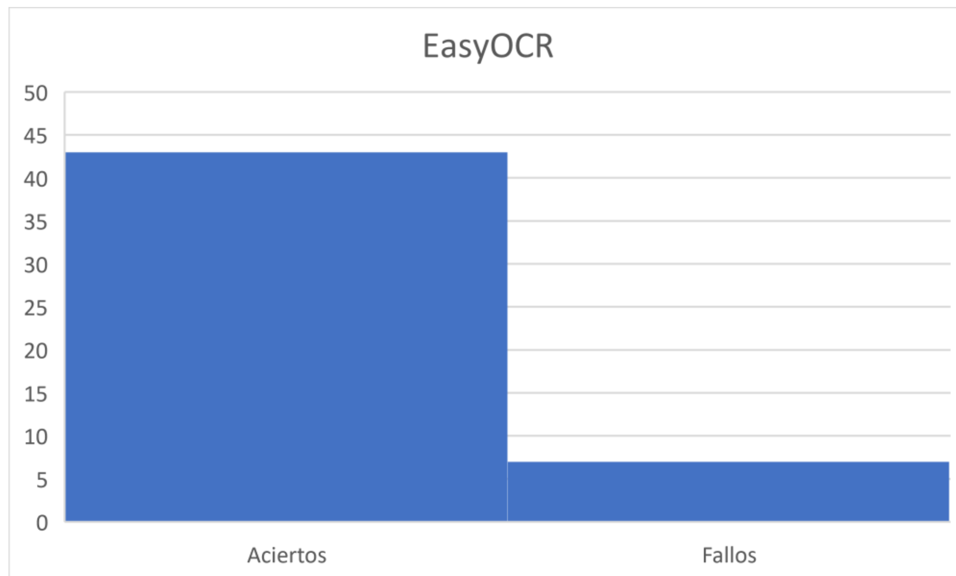


Figura 142. Aciertos en detecciones de EasyOCR para superficie de metal liso (43/50 aciertos). Fuente: Elaboración propia.

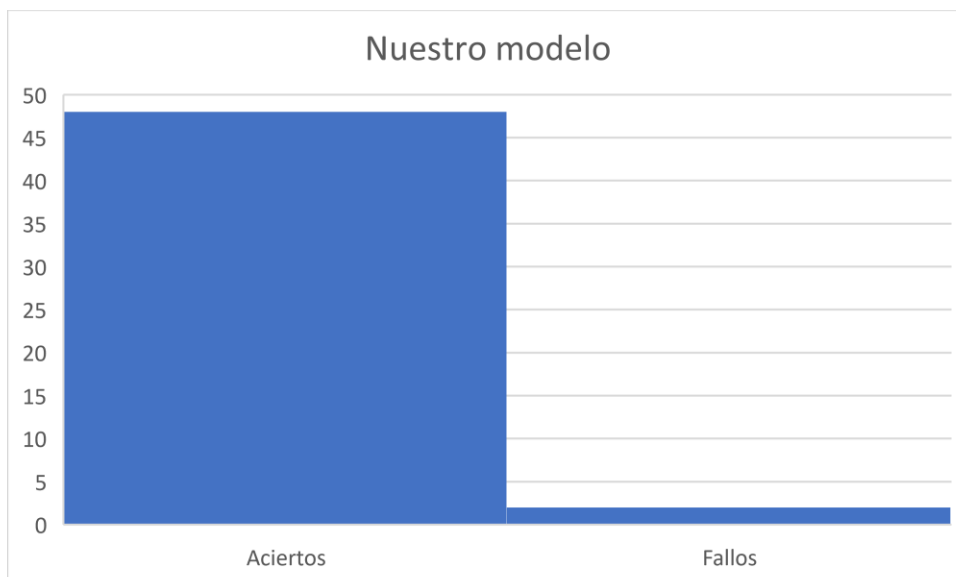


Figura 143. Aciertos en detecciones de nuestro modelo para superficie de metal liso (48/50 aciertos). Fuente: Elaboración propia.

Finalmente, podemos observas como con las 20 imágenes con superficie de metal liso, ambos modelos han tenido unos buenos resultados.

En este caso nuestro modelo ha acertado 48 imágenes mientras que EasyOCR ha acertado 43 del total de 50 fotos evaluadas.

14. CONCLUSIÓN

En conclusión, a través del entrenamiento de un modelo con un dataset generalista (formado por 4 superficies industriales diferentes), hemos logrado demostrar que supera al reconocido modelo **EasyOCR** en todos los ámbitos en los que ha sido evaluado. Este resultado demuestra la importancia de la calidad y la diversidad de los datos utilizados durante el proceso de entrenamiento, ya que son fundamentales para obtener un modelo altamente preciso y eficiente en el reconocimiento óptico de caracteres.

Este logro refuerza la importancia de la adaptabilidad y personalización de los modelos de aprendizaje automático.

Aunque **EasyOCR** es un modelo reconocido y ampliamente utilizado, su estructura de CRNN (redes neuronales convolucionales recurrentes) en este caso no le beneficia, haciendo relativamente sencillo que un modelo estructurado únicamente con CNN (redes neuronales convolucionales) con pocas imágenes pueda superarlo.

15. PRÓXIMOS PASOS

Después de demostrar lo esencial que es un dataset variado para que un entrenamiento sea efectivo, el siguiente paso será generar un dataset aún más grande con una variedad mucho más amplia para que el modelo aun pueda ser sometido a muchas más superficies.

Esto implica recolectar imágenes de diferentes entornos industriales (a esto nos ayudarán los proyectos que vayan saliendo en la empresa de temática OCR), capturar distintas condiciones de iluminación, tamaños de texto. Además, sería interesante considerar la generación de datos sintéticos para aumentar aún más la variabilidad y diversidad del dataset con los códigos explicados en este proyecto. Esto permitirá que el modelo se adapte de manera más precisa a diferentes superficies industriales y mejore su capacidad de reconocimiento en escenarios del mundo real.

De cara a la generalización del modelo en otras superficies y alfabetos a los tratados anteriormente, se podría llegar a conseguir añadiendo imágenes de códigos sobre distintas superficies como plástico, goma, etc.… y con la adición de otro tipo de alfabeto distinto al anglosajón, con ello conseguiríamos un modelo aún más generalista del que hemos conseguido.

El principal problema a la hora de realizar esta tarea es la gran cantidad de tiempo que requiere, ya que, como se ha mencionado anteriormente, apenas hay datasets útiles para el OCR.

A parte de la detección del código, también se podría llegar a evaluar la calidad del grabado de dicho código sobre la superficie que corresponda, llegando incluso, a detectar problemas en los sistemas de impresión de las industrias.

16. PRESUPUESTO

A continuación, se van a detallar los materiales necesarios para la ejecución de un proyecto general para el reconocimiento óptico de caracteres (OCR).

Para este ejemplo hemos escogido componentes de gama media, ya que únicamente contamos con un modelo de YOLO, así como una única cámara

Componente	Precio/Ud.	Unidades	Total
CPU (i7-11700F) / 16gb RAM 1TB SSD / GPU NVIDIA RTX3060	1.000,69 €	1	1.000,69 €
Cable ethernet RJ45 CAT 8	20,00 €	2	40,00 €
Cámara ER-630-018GC PoE	335,07 €	1	335,07 €
Lente LCM-5MP-25MM-F2.0-1.8-ND1 C-MOUNT	84,00 €	1	84,00 €
Iluminación LED1-BL White	134,00 €	1	134,00 €
Switch gestionable D-Link DGS-1100-08PV2	113,75 €	1	113,75 €
Monitor Asus VT229H 21.5" LED IPS FullHD Táctil	253,99 €	1	253,99 €
Disco duro externo Seagate 1 TB HDD	53,99 €	1	53,99 €
Dimmer 12V 24V 30A Controlador de regulador para iluminación	14,99 €	1	14,99 €
Fuente de alimentación Meanwell MDR-60-24 AC-DC	23,03 €	1	23,03 €
		Total	2.053,51 €

17. BIBLIOGRAFÍA

1. Abdelkrim Alahyane, M. E. (2021). *Open data for Moroccan license plates for OCR.* Marruecos: Arxiv.
2. Abhishek Bamotra, P. K. (2023). *TransDocs: Optical Character Recognition with word.* Pittsburgh: Arxiv.
3. Ali Furkan Biten, R. T. (2022). *CR-IDL: OCR Annotations for Industry.* Arxiv.
4. Baoguang Shi, X. B. (2015). *An End-to-End Trainable Neural Network for Image-based Sequence Recognition and Its Application to Scene Text Recognition.* Wuhan: Arxiv.
5. Celine Thillou, S. F. (2005). *An Embedded Application for Degraded Text Recognition.* Bélgica: 2005 Hindawi Publishing Corporation.
6. J. P. Naiman, P. K. (2022). *Figure and Figure Caption Extraction for Mixed Raster and Vector PDFs: Digitization of Astronomical Literature with OCR Features.* Urbana-Champaign: Arxiv.
7. JaidedAI. (s.f.). *Github/EasyOCR*. Obtenido de https://github.com/JaidedAI/EasyOCR
8. Kataria, B. (30 de Marzo de 2019). *CNN-Bidirectional LSTM Based Optical Character Recognition of Sanskrit.* Patan: ISSN: 2456-3307. Obtenido de https://ijsrcseit.com/CSEIT2064126
9. Khan, N. H. (16 de Agosto de 2018). *Urdu Optical Character Recognition Systems: Present Contributions and Future Directions.* Pakistan: IEEE 2169-3536 . Obtenido de https://ieeexplore.ieee.org/document/8438450
10. Nelson, J. (6 de Enero de 2020). *Roboflow*. Obtenido de https://blog.roboflow.com/why-preprocess-augment/#:~:text=Image%20preprocessing%20is%20the%20steps,and%20increase%20model%20inference%20speed.
11. Noman Islam, Z. I. (2016). *A Survey on Optical Character Recognition System.* ISSN-2409-6520.
12. Samet Bayram, K. B. (2022). *A Black-Box Attack on Optical Character.* Newark: Arxiv.
13. Yang, H. (27 de Febrero de 2014). *ieeexplore.ieee.org*.
14. Youngmin Baek, B. L. (2019). *Character Region Awareness for Text Detection.* Seongnam-si: Arxiv.